# On-Chip Learning and Inference Acceleration of Sparse Representations

by

Deepak Vinayak Kadetotad

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved May 2019 by the
Graduate Supervisory Committee:

Jae-sun Seo, Chair
Yu (Kevin) Cao
Chaitali Chakrabarti
Sarma Vrudhula

ARIZONA STATE UNIVERSITY

August 2019

ABSTRACT

The past decade has seen a tremendous surge in running machine learning (ML) functions on mobile devices, from mere novelty applications to now indispensable features for the next generation of devices. While the mobile platform capabilities range widely, long battery life and reliability are common design concerns that are crucial to remain competitive. Consequently, state-of-the-art mobile platforms have become highly heterogeneous by combining a powerful CPUs with GPUs to accelerate the computation of deep neural networks (DNNs), which are the most common structures to perform ML operations. But traditional von Neumann architectures are not optimized for the high memory bandwidth and massively parallel computation demands required by DNNs. Hence, propelling research into non-von Neumann architectures to support the demands of DNNs.

The re-imagining of computer architectures to perform efficient DNN computations requires focusing on the prohibitive demands presented by DNNs and alleviating them. The two central challenges for efficient computation are (1) large memory storage and movement due to weights of the DNN and (2) massively parallel multiplications to compute the DNN output.

Introducing sparsity into the DNNs, where certain percentage of either the weights or the outputs of the DNN are zero, greatly helps with both challenges. This along with algorithm-hardware co-design to compress the DNNs is demonstrated to provide efficient solutions to greatly reduce the power consumption of hardware that compute DNNs. Additionally, exploring emerging technologies such as non-volatile memories and 3-D stacking of silicon in conjunction with algorithm-hardware co-design architectures will pave the way for the next generation of mobile devices.

Towards the objectives stated above, our specific contributions include (a) an architecture based on resistive crosspoint array that can update all values stored and

compute matrix vector multiplication in parallel within a single cycle, (b) a framework of training DNNs with a block-wise sparsity to drastically reduce memory storage and total number of computations required to compute the output of DNNs, (c) the exploration of hardware implementations of sparse DNNs and architectural guidelines to reduce power consumption for the implementations in monolithic 3D integrated circuits, and (d) a prototype chip in 65nm CMOS accelerator for long-short term memory networks trained with the proposed block-wise sparsity scheme.

*Dedicated to my parents:*

*Manjula Kadetotad and Vinayak Kadetotad*

# ACKNOWLEDGEMENTS

My deepest gratitude is to my advisor, Dr. Jae-sun Seo, for the patience, advice and guidance he has offered throughout my study. His insightful recommendations helped me navigate through challenges that I faced during my study. In particular, his tips about software development, writing papers, attending conferences, time management and networking have been extremely useful. This has led to one of the most productive times in my life. It is natural that this dissertation would not be possible without his support.

I am thankful to Dr. Chaitali Chakrabarti, Dr. Sarma Vrudhula, and Dr. Yu Cao for taking out time and being in my Ph.D. defense committee.

I would also like to thank my colleagues who worked and collaborated with me during the entirety of my Ph.D. journey.

Finally, my parents Manjula Kadetotad and Vinayak Kadetotad. I am grateful for their love, endless support and understanding towards my research career.

TABLE OF CONTENTS

Page

LIST OF TABLES ..... ix

LIST OF FIGURES ..... x

CHAPTER

1 INTRODUCTION ..... 1

1.1 Motivation and Challenges ..... 3

1.1.1 In-memory Computing Architecture ..... 3

1.1.2 Hardware-Algorithm Co-design for Hardware Accelerators .. 4

1.2 Dissertation Outline ..... 5

2 PARALLEL ARCHITECTURE WITH RESISTIVE CROSSPOINT ARRAY FOR DICTIONARY LEARNING ACCELERATION ..... 7

2.1 Introduction ..... 7

2.2 RRAM Device ..... 12

2.3 Crosspoint Array Architecture and Design ..... 15

2.3.1 Overall Architecture of PARCA ..... 15

2.3.2 Read: Integrate and Fire ..... 19

2.3.3 Write: Timing based Local Programming ..... 22

2.4 Simulation Results ..... 25

2.4.1 Simulations with single RRAM device ..... 26

2.4.2 Simulations with 100×100 RRAM array ..... 27

2.4.3 Quantization Error ..... 30

2.4.4 Sneak Path and interconnect in RRAM ..... 31

2.4.5 Variability and Mismatch ..... 33

2.5 Demonstration in Learning ..... 34

2.6 Conclusion ..... 36

CHAPTER Page

3 EFFICIENT MEMORY COMPRESSION IN DEEP NEURAL NETWORKS USING COARSE-GRAIN SPARSIFICATION FOR SPEECH APPLICATIONS ..... 37
3.1 Introduction ..... 38
3.2 DNN Processing in Speech Applications ..... 41
3.2.1 DNNs for Keyword Detection and Speech Recognition ..... 41
3.2.2 Training DNNs ..... 42
3.2.3 Feedforward Classification of DNNs ..... 43
3.3 DNN Weight Compression Using Sparsity and Precision Reduction ..... 44
3.3.1 Coarse-Grain Sparsification (CGS) ..... 44
3.3.2 Low-Precision Classification ..... 46
3.4 Hardware Architecture and CMOS Implementation ..... 47
3.4.1 Keyword Detection DNN Implementation ..... 47
3.4.2 Speech Recognition DNN Implementation ..... 48
3.4.3 Finite State Machine ..... 49
3.5 Experimental Results ..... 49
3.5.1 Experimental Setup ..... 50
3.5.2 Keyword Detection DNN ..... 51
3.5.3 Speech Recognition DNN ..... 56
3.6 Conclusion ..... 60
4 POWER, PERFORMANCE, AND AREA BENEFIT OF MONOLITHIC 3D ICs FOR ON-CHIP SPARSE DEEP NEURAL NETWORKS TARGETING SPEECH RECOGNITION ..... 61
4.1 Introduction ..... 61

CHAPTER Page

4.2 Deep Neural Network for Speech Recognition . . . . . 64
4.2.1 Our DNN Topology . . . . . 64
4.2.2 DNN Training and Classification . . . . . 65
4.2.3 Coarse-Grain Sparsification (CGS) . . . . . 66
4.3 Full-Chip Monolithic 3D IC (M3D) Design Flow . . . . . 68
4.4 DNN Architecture Description . . . . . 70
4.5 M3D Impact on Energy-Efficiency . . . . . 72
4.5.1 Area, Wirelength, and Capacitance Comparisons . . . . . 73
4.5.2 Power Comparisons . . . . . 75
4.6 M3D Impact on Performance . . . . . 77
4.7 Architectural Impact Discussions . . . . . 81
4.7.1 CGS-16 vs. CGS-64 Architecture Comparisons . . . . . 81
4.7.2 Impact of Workloads . . . . . 84
4.8 Observations and Guidelines . . . . . 86
4.9 Conclusions . . . . . 87
5 HCGS: COMPRESSING LSTM NETWORKS WITH HIERARCHICAL COARSE-GRAIN SPARSITY . . . . . 92
5.1 Introduction . . . . . 92
5.2 Related Work . . . . . 95
5.3 HCGS Based LSTM Training . . . . . 97
5.3.1 Long Short-Term Memory RNN . . . . . 97
5.3.2 Hierarchical Coarse-Grain Sparsity (HCGS) . . . . . 99
5.3.3 Quantizing LSTM Networks . . . . . 102
5.4 Experiments . . . . . 103

CHAPTER Page

5.4.1 Experimental Setup . . . 103
5.4.2 HCGS Robustness to Block Size . . . 105
5.4.3 Improvements due to HCGS . . . 106
5.4.4 LSTM Results for TIMIT . . . 108
5.4.5 LSTM Results for TED-LIUM . . . 109
5.5 Discussion . . . 111
5.6 Conclusion . . . 112
6 A 8.93 TOPS/W LSTM RECURRENT NEURAL NETWORK ACCELERATOR FEATURING HIERARCHICAL COARSE-GRAIN SPARSITY WITH ALL PARAMETERS STORED ON-CHIP . . . 115
6.1 Introduction . . . 115
6.2 LSTM and Hierarchical Coarse-Grain Sparsity . . . 116
6.2.1 LSTM-based Speech Recognition . . . 116
6.2.2 Hierarchical Coarse-Grain Sparsity . . . 118
6.3 Architecture and Design Optimizations . . . 119
6.3.1 Hardware Architecture . . . 119
6.3.2 Interleaved Memory Storage . . . 121
6.3.3 Design Space Exploration . . . 122
6.4 Measurement & Comparison . . . 123
6.5 Conclusion . . . 125
7 CONCLUSION . . . 129
REFERENCES . . . 131

LIST OF TABLES


Table Page

2.1 PARCA Operations for Key Sparse Coding Tasks. . . . . 18

2.2 Evaluation of the speedup in computing and energy. . . . . 35

3.1 Comparison of memory requirements and AUC for keyword detection networks. . . . . 53

3.2 Power and area for keyword detection networks. . . . . 56

3.3 Comparison of memory, SER, and WER for speech recognition networks. 58

3.4 Evaluation of the speedup in computing and energy. . . . . 60

4.1 Key parameters of the two CGS-based DNN architectures used in our study: block size of 16×16 (DNN CGS-16) and block size of 64×64 (DNN CGS-64). . . . . 68

4.2 Iso-performance (400MHz) comparison of design metrics of 2D and M3D designs of DNN CGS-16 and DNN CGS-64 architectures. All percentage values show the reduction from their 2D counterparts. . . . . 89

4.3 Iso-performance (400MHz) power comparison of two architectures (CGS-16 vs. CGS-64) using two workloads (classification vs. pseudo-training). All percentage values show the reduction from their 2D counterparts. . . 90

4.4 Maximum performance comparison of 2D and M3D designs of CGS-16 and CGS-64 architectures. . . . . 91

4.5 Key parameter comparison of the worst timing path in Fig. 4.10 of the 2D and M3D designs of DNN CGS-16 architecture. . . . . 91

5.1 Comparison of different LSTM network configurations that lead to the same memory storage requirements. . . . . 104

6.1 Comparison of RNN performance with prior works. . . . . 126

LIST OF FIGURES

1.1 RRAM crosspoint array with proposed peripheral circuits for in-memory computing . . . 3

1.2 Illustration of coarse-grain weight compression for DNN inference acceleration. . . . 5

Figure Page

2.1 Similarity of biological neural network and the RRAM crosspoint array, in network structure, device plasticity, and local programming. (a) Plasticity of biological synapses governed by local neuron spikes. (b) RRAM based crosspoint array. . . . 8

2.2 Sub-circuit module of a single resistive cell ($S$ : cell spacing; $W$ : wire width). The cell capacitor ($C_r$) is in parallel with the cell resistor ($R_r$). The wire resistors ($R_w$) and capacitors ($C_w$) for top and bottom interconnect are considered. Sub-circuit is duplicated in simulations for the array. . . . 13

2.3 Experimental data Park *et al.* (2013) of conductance modulation in RRAM based synapse in linear scale is shown along with the best fit graph of the raw data. . . . 14

2.4 PARCA architecture with peripheral Read and Write modules. $Z$ and $X$ (or $r$) nodes have the same Read (Section 2.3.2), but different Write circuits (Section 2.3.3). All RRAM cells perform Read or Write operation in parallel. . . . 16

2.5 Procedure to obtain the final $D \cdot Z$ value from read out values of single bits. The final operation involves encoding the spike counts to binary values and summing them. . . . 17

Figure Page

2.6 Schematic of the single-ended adaptive Read circuit is shown. Based on the IF neuron model, it converts an analog input current $I_{r,i}$ into a digital number. . . . . 19

2.7 Effect of ATB on single-ended adaptive Schmitt trigger based read circuit. . . . . 20

2.8 Schematics of the differential Read circuit. Latency and sensitivity to variation is improved compared to the single-ended design. . . . . 21

2.9 Write circuit for $Z$, which generates a timing window that is proportional to $Z$. . . . . 23

2.10 Write circuit for $r$, which generates a series of evenly-spaced spikes whose firing rate is proportional to $r$. . . . . 24

2.11 The operation of the single-ended read circuit for two input currents: (left) $I_r = 5.75\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1. . . . . 25

2.12 The operation of the differential read circuit for two input currents: (left) $I_r = 5.75\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1. . . . . 26

2.13 The operation of the single-ended read circuit for two input currents: (left) $I_r = 6\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1. . 27

2.14 The operation of the differential read circuit for two input currents: (left) $I_r = 5.75\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1. . . . . 28

2.15 The overlap in time between $Z$ and $r$ pulses modulates $D$. . . . . 28

Figure Page

2.16 100×100 array simulation of differential read circuits for two input currents: (left) $I_r = 5.55\mu A$; and (right) $I_r = 1.25\mu A$; the corresponding $n_i$ is 6 and 1. . . . . 29
2.17 100×100 array simulation for write operation by overlapping $Z$ and $r$ pulses. . . . . 30
2.18 Quantization of read and write circuits are shown. (a) Number of pulses and RRAM current show a close-to-linear relationship. (b) Digitally programmed pulse width closely follows the mathematical multiplication. . . . . 31
2.19 Sneak paths could cause current to flow through inactive nodes during the read or write operation. . . . . 32
2.20 Results of Monte Carlo simulations which measure the number of pulses against the total count. Both instances correspond to 1 pulse being the correct output. (a) Adaptive read circuit, (b) Differential read circuit. . 33
2.21 Demonstration of dictionary learning with MNIST data. . . . . 34
3.1 DNNs for (a) keyword detection and (b) speech recognition. . . . . 42
3.2 DNNs for (a) keyword detection and (b) speech recognition. . . . . 44
3.3 Block diagram of the DNN-based classification system. . . . . 47
3.4 Effect of block size and percentage drop on average AUC of keyword detection DNN. . . . . 52
3.5 ROC Curve of different implementations for keyword detection DNN. . . 54
3.6 For the same real-time performance, supply voltage, system power, and area results for keyword detection DNN are shown for different number of parallel MAC units in (a) 65nm LP and (b) 40nm LP CMOS. . . . . 55

Figure Page

3.7 For the same real-time performance, supply voltage, system power, and area results for keyword detection DNN are shown for different number of parallel MAC units in (a) 65nm LP and (b) 40nm LP CMOS. ..... 57
3.8 For the same real-time performance, supply voltage, system power, and area results for keyword detection DNN are shown for different number of parallel MAC units in (a) 65nm LP and (b) 40nm LP CMOS. ..... 59
4.1 A schematic showing a gate-level monolithic 3D IC (M3D). ..... 63
4.2 Diagram of our DNN for speech recognition. ..... 65
4.3 1024×1024 weight matrix is divided into 64×64 weight blocks with each weight block having 16×16 weights (i.e. block size of 16×16). 87.5% of weight blocks are dropped using coarse-grain sparsification (CGS). The remaining 12.5% weight blocks are stored in memory. ..... 68
4.4 Block diagram of the proposed CGS-based DNN architecture for speech recognition. ..... 70
4.5 28nm full-chip layouts of DNN CGS-16 and CGS-64 architectures at 400MHz target clock frequency. (a) 2D IC design, (b) M3D design with memory blocks on both tier (M3D-both), (c) M3D design with memory blocks on a single tier (M3D-one), (d) 2D IC, (e) M3D-both, (f) M3D-one. ..... 72
4.6 Cell placement of the modules in CGS-16 architecture. (a) 2D, (b) M3D-both, (c) M3D-one. Each module is highlighted with different colors. ..... 75
4.7 Wirelength distribution of CGS-16 architecture. ..... 77
4.8 Cell drive-strength distribution of CGS-16 architecture. ..... 78

4.9 Full-chip die images of 2D and M3D designs of DNN CGS-16 and CGS-64 architectures at their maximum target clock frequency. (a) 2D design at 550MHz, (b) M3D design at 575MHz of DNN CGS-16 architecture, (c) 2D design at 600MHz, (d) M3D design at 625MHz of DNN CGS-64 architecture. . . . . 78

4.10 Worst timing path comparison of 2D and M3D designs of CGS-16 architecture. (a) The worst timing path of 2D design at its maximum target clock frequency, 550MHz. (b) The same timing path in M3D design. Cells in the top tier are projected into the bottom tier for M3D design, and red boxes (i.e., weight SRAM) indicate the start point, whereas blue boxes (i.e., flip-flops in MAC unit) represent the end point of the timing path. Yellow lines show the wires in 2D and the bottom tier of M3D design, whereas green lines are the top tier wires in M3D design. . . . . 80

4.11 Slack distribution comparison between 2D and M3D designs of DNN CGS-16 architecture at the maximum clock frequency of the M3D design. 81

4.12 Standard cell area breakdown of 2D CGS-16 and 2D CGS-64 architectures. Non-dashed and dashed boxes respectively indicates combinational and sequential elements. Only five largest modules are shown. . . . 82

4.13 Power breakdown under two architectures (CGS-16 vs. CGS-64), two workloads (classification vs. pseudo-training), and two designs (2D vs. M3D). . . . . 83

Figure Page

4.14 Comparison of (a) the total wirelength and (b) the total cell count of the timing paths from weight SRAMs to registers in MAC units through neuron selection logic of 2D and M3D-both designs of CGS-16 and CGS-64 architecture. . . . . 84

4.15 Comparison between (a) the feed-forward classification and (b) pseudo-training . . . . . 85

5.1 Comparison of index and weight memory requirement among three compression methodologies for 4×, 8×, and 16× compression. (a) Compression of 8-bit weights. (b) Compression of 4-bit weights. . . . . 93

5.2 LSTM computation flow for each layer. Each of the four gates of the LSTM layer receives the input sequence $x_t$ and the recurrent hidden state sequence $h_{t-1}$, along with the corresponding weights and biases. . . 98

5.3 Proposed hierarchical block-wise compression of weights. . . . . 99

5.4 Further reduction of index memory aided by sharing the random connection mask for four gates in each LSTM layer. . . . . 101

5.5 PER (TIMIT) and WER (TED-LIUM) values are shown for LSTMs trained with different HCGS block sizes. For all datapoints, compression rate is 16× and weight precision is 6-bit. . . . . 105

5.6 PER (TIMIT) comparison between single-tier CGS and two-tier HCGS schemes. . . . . 106

5.7 Network convergence comparison between various different target compression. . . . . 107

Figure Page

5.8 PER vs. RNN weight memory results for different sizes of 2-layer LSTMs for TIMIT, with various compression rates and quantization values. . . . . 109
5.9 PER and weight memory are shown for prior LSTM compression works and the Pareto frontier curve obtained with HCGS-based LSTMs. . . . . 110
5.10 WER vs. RNN weight memory results for different sizes of 3-layer LSTMs for TED-LIUM, with various compression rates and quantization values. . . . . 111
6.1 Illustration of LSTM cell with computation equations. . . . . 117
6.2 Illustration of LSTM RNN weight compression featuring the proposed hierarchical coarse-grain sparsity (HCGS). . . . . 118
6.3 Overall architecture of the proposed LSTM RNN accelerator. . . . . 119
6.4 LSTM data flow and core computations. . . . . 122
6.5 HCGS design space exploration. (a) RNN width and the number of CGS levels. (b) HCGS block size and random block selection. . . . . 123
6.6 Prototype chip micrograph and performance summary. . . . . 124
6.7 Power and frequency measurement results with voltage scaling for (a) 2-layer LSTM for TIMIT and (b) 3-layer LSTM for TED-LIUM. (c) Measurement results for energy-efficiency (TOPS/W) and leakage power. (d) Power breakdown of 3-layer LSTM at 0.75V supply. . . . . 127
6.8 Comparison of TIMIT PER and energy efficiency (frames per second/power, FPS/W) with prior LSTM implementations. . . . . 128

# Chapter 1

# INTRODUCTION

As the application of Deep Neural Networks (DNNs) have become ubiquitous in many computing fields, the demand for efficient neural network accelerators has increased considerably. DNNs started initially as fully connected Multi Layer Perceptrons (MLP) Rosenblatt (1961) and trained with the back propagation algorithm Rumelhart *et al.* (1985). MLPs were more successful with classification problems when compared to regression based approaches, but they were often over-parameterized. Variations in DNNs appeared and are optimized to share weights (1) spatially, such as Convolutional Neural Networks (CNNs) for computer vision Krizhevsky *et al.* (2012) and (2) temporally, such as Recurrent Neural Networks (RNNs) on natural language processing Robinson *et al.* (1996).

DNNs take advantage of the rapidly improved computation structure to learn from very larger training datasets, leading to exceptional recognition accuracy close to or even better than human-level perception. This was a stark improvements from traditional machine learning algorithms that relied on hand-crafted features from signal or data processing algorithms. DNNs with significantly improved accuracy and expanded application domains came with computation bottlenecks, where the requirement of massively parallel operations and memory bandwidth required for the computation, still challenge the state-of-art computing platforms to achieve real-time performance with high energy efficiency.

To realize high throughput, high performance GPUs are often used to accelerate the training and inference tasks of DNNs, as they can take advantage of the thousands of parallel cores, operating at high clock frequencies at GHz level, and achieve

hundreds of GB/s memory bandwidth. However, their power consumption is too high (>150W) for power and energy constrained platforms. Furthermore, GPUs are best suited for achieving high throughput when processing large batches of images. However, for applications that require very low latency for processing a single image, as in autonomous drive and surveillance, the completion of detection must be done at the speed of incoming data stream, which degrades GPUs' performance and energy-efficiency substantially.

On the other hand, various deep learning hardware accelerators have been recently proposed based on application specific integrated circuits (ASICs) Lee *et al.* (2019), system on chips (SoCs) Gokhale *et al.* (2014) and field-programmable gate arrays (FPGAs) Wang *et al.* (2019) targeting at high performance and energy efficiency.

ASICs have gained increasing interests and popularity in particular to accelerate the inference tasks, due to their (1) superior energy efficiency compared to GPUs and FPGAs Shin *et al.* (2017); Lee *et al.* (2019) (2) througput performance. The high performance and efficiency of an ASICs can be realized by synthesizing a circuit that is customized for a specific computation to directly process the operations with customized memory systems. However, energy efficiency for implementing DNN accelerators using either conventional architectures or vanilla DNN algorithms will not be sufficient for mobile devices. Therefore, an effort into Hardware-Algorithm co-design will be required to boost the energy efficiency of DNN hardware accelerators to successfully and cost-effectively be implemented in mobile devices.

The goal of this dissertation is to explore new algorithms and architectures for hardware accelerators with high performance and energy efficiency, especially for networks or methods with sparse representations.

## 1.1 Motivation and Challenges

State of the art DNNs currently have millions of parameters required for storage and movement of those parameters to and from DRAMs is extremely restrictive to achieving the best energy efficiency possible Chen *et al.* (2017). To that effect, either reducing the total data to be transferred or combining storage and computation of data into a single system can be provided as solutions.

### *1.1.1 In-memory Computing Architecture*

In-memory computing combines the storage of weights and computation to be performed on the weights into a hybrid structure. The reduced separation between storage and computation improves energy efficiency greatly and has been proposed as a promising solution for learning in hardware neural networks Afifi *et al.* (2010); Rajendran *et al.* (2013). At each cross point, the conductance ($G$) of a memory cell represents weight value.

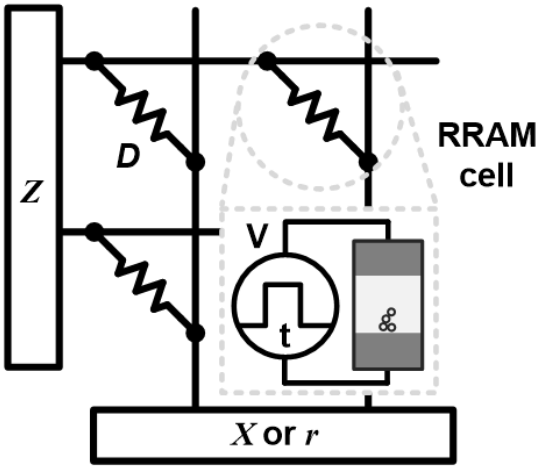



**Figure 1.1:** RRAM crosspoint array with proposed peripheral circuits for in-memory computing

The two core functions proposed to implement using the resistive crosspoint array include:

1. *Read for Matrix-Vector Multiplication*: When a voltage is input from $Z$, the output current at $x_i$ is $I_{X,i} = \sum G_{ij} \cdot V_{Z,j}$. If $G$ encodes the weight value, then a Read corresponds to sensing the current which encodes $D \cdot Z$, which is computed in parallel.

2. *Write to Update Weights*: Using the RRAM crosspoint the conductance of the entire array can updated in parallel. Previous approaches involve sequential operations (row-by-row, column-by-column, or even bit-by-bit) to update $G$ of the RRAM cell.

The high level of parallelism in combination with the non-volatile RRAM crosspoint array of the proposed architecture can potentially allow for on-chip learning and inference acceleration in mobile devices.

### *1.1.2 Hardware-Algorithm Co-design for Hardware Accelerators*

Weights can be stored on-chip (e.g. SRAM cache of mobile processors), which has fast access time (nanoseconds range), but is limited to a few mega bytes (MB) due to cost Halpern *et al.* (2016). Alternatively, weights can be stored off-chip (e.g. DRAM) up to a few gigabytes (GB), but access is slower (tens of nanoseconds range) and consumes $\sim$100$\times$ higher energy than on-chip counterparts Han *et al.* (2015b).

To improve energy-efficiency of DNN hardware, off-chip memory access and communication need to be minimized Chen *et al.* (2017). To that end, it becomes crucial to store most or all weights on-chip through sparsity/compression, weight quantization, and network size reduction. But this objective can only be met through hardware-algorithm co-design.

DNN model compression through element-wise sparsity can result in a large compression of DNN weights Han *et al.* (2015a), but the index storage can be as large

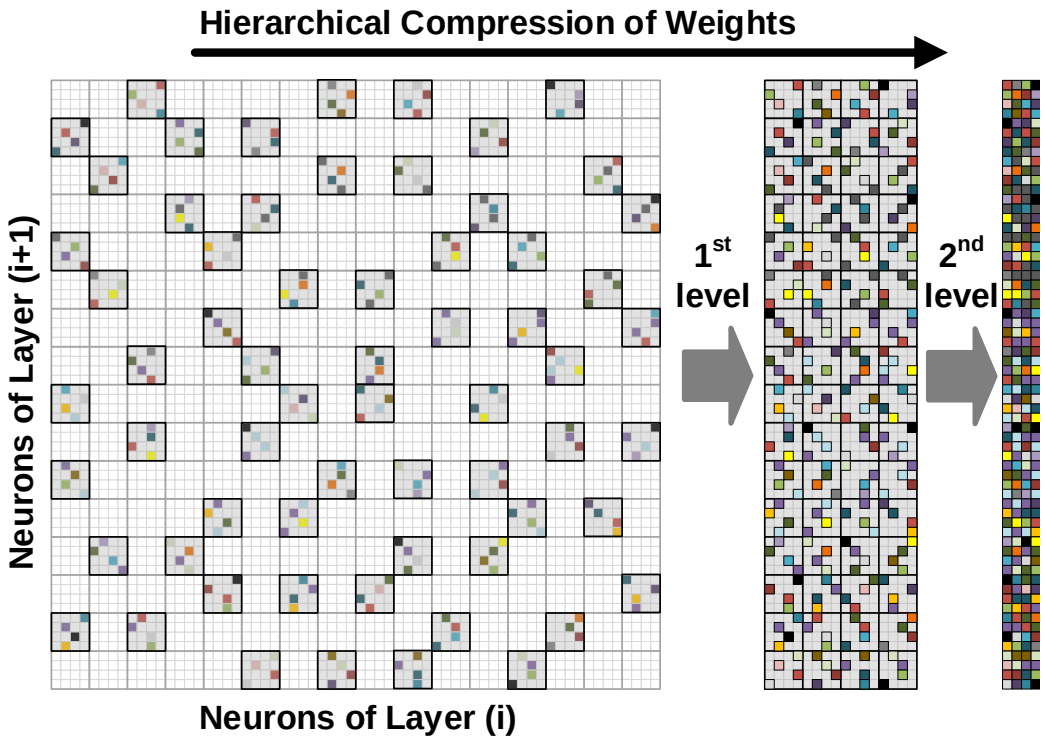



**Figure 1.2:** Illustration of coarse-grain weight compression for DNN inference acceleration.

as the non-zero weights themselves, especially if we use the simple coordinate (COO) format that stores the location of each non-zero weight. The compressed sparse row (CSR) or compressed sparse column (CSC) format Han *et al.* (2017) reduces the index cost, but still exhibit noticeable index memory and causes irregular memory access Wang *et al.* (2018).

Structured or coarse-grain compression as in Fig 1.2 is therefore proposed, which minimizes the index storage, makes memory access more regular, and enhances DNN inference acceleration.

## 1.2 Dissertation Outline

In this dissertation a complete framework for on-chip learning using a novel RRAM crosspoint architecture and a hardware-algorithm co-design DNN inference accelera-

tor is proposed. The chapters are organized as follows:

- Chapter 2 describes the authors work related to the accelerator design based on a resistive crosspoint array, where the novel design of peripheral circuits in conjunction with emerging resistive non-volatile memory (RRAM) leads up to 3000× acceleration of implementation of sparse coding.

- Chapter 3 outlines the introduction of coarse grain sparsification (CGS), which is a novel method to sparsify weight connections between layers of a neural network (NN) in a block-wise manner. This greatly improved efficiency hardware implementations of NNs.

- Chapter 4 details the experiments conducted to explore the effects of implementing CGS on a varied set of parameters that will affect the power consumption, area and performance of the design hardware accelerator on the emerging monolithic 3D integrated circuit technology.

- Chapter 5 describes the further evolution of the CGS scheme, where the block-wise sparsity is implemented in a hierarchical manner in a method called hierarchical coarse grain sparsification (HCGS). Experiments conducted to explore different combinations of block sizes and the co-relation between sparsity and accuracy are detailed in the chapter as well.

- Chapter 6 corresponds to the architectural design of a long-short term memory accelerator for speech recognition based on the HCGS compression scheme. The chapter details the advantages of HCGS for hardware design of NN accelrators and the measurements of the 65nm prototype chip.

- Chapter 7 amalgamates the ideas provided above with a description of the underlying theme of the entire thesis and the conclusions arrived by the author.

Chapter 2

# PARALLEL ARCHITECTURE WITH RESISTIVE CROSSPOINT ARRAY FOR DICTIONARY LEARNING ACCELERATION

This chapter proposes a parallel architecture with resistive crosspoint array. The design of its two essential operations, Read and Write, is inspired by the biophysical behavior of a neural system, such as integrate-and-fire and local synapse weight update. The proposed hardware consists of an array with resistive random access memory (RRAM) and CMOS peripheral circuits, which perform matrix-vector multiplication and dictionary update in a fully parallel fashion, at the speed that is independent of the matrix dimension. The read and write circuits are implemented in 65nm CMOS technology and verified together with an array of RRAM device model built from experimental data. The overall system exploits array-level parallelism and is demonstrated for accelerated dictionary learning tasks. As compared to software implementation running on a 8-core CPU, the proposed hardware achieves more than 3000$\times$ speedup, enabling high-speed feature extraction on a single chip.

## 2.1 Introduction

The biophysical neural system has been a rich source of inspiration for computing beyond the conventional von Neumann architecture. By connecting a massive number of spiking neurons through synapses, our brain learns how to recognize various objects and make decisions. It is also hypothesized that training is achieved through plastic synapses, which change their weights based on the spike timing of pre-synaptic and post-synaptic neuron. This learning rule is known as spike-timing-dependent-plasticity (STDP) Song *et al.* (2000); Bi and Poo (1998) (Fig. 2.1(a)). Cognitive

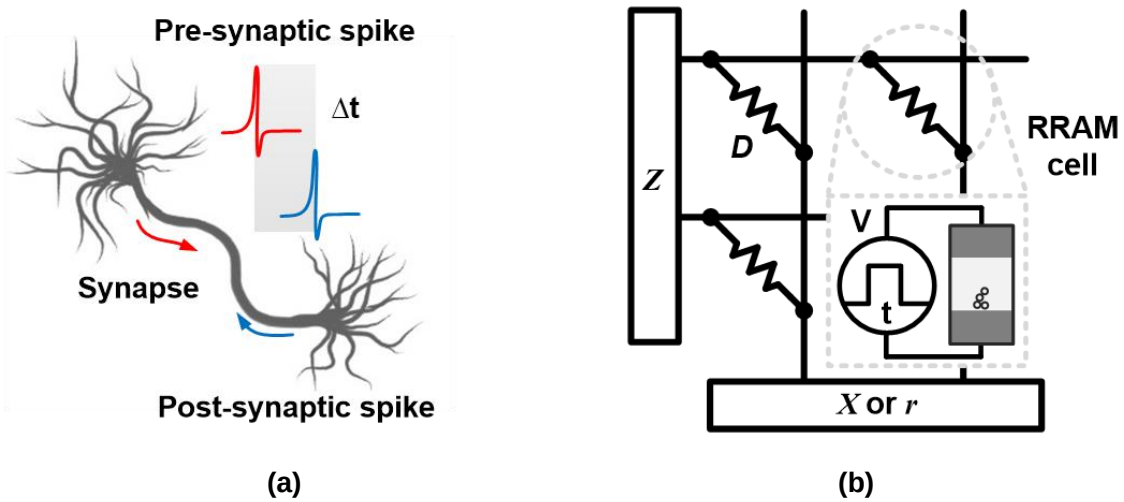



**Figure 2.1:** Similarity of biological neural network and the RRAM crosspoint array, in network structure, device plasticity, and local programming. (a) Plasticity of biological synapses governed by local neuron spikes. (b) RRAM based crosspoint array.

tasks such as learning and classification have also been pursued by a number of machine learning algorithms. Among them, sparse coding is a widely used unsupervised learning algorithm for audio processing and image recognition Olshausen and Field (1996) as well as representation of the visual cortex Tosic and Frossard (2011). It aims to represent input vectors with sparse linear combinations of basis elements in a dictionary. We adopt the commonly used L1-norm minimization Tosic and Frossard (2011) as the objective function of sparse coding as follows:

$$\sum_i \|D \cdot Z_i - x_i\|^2 + \lambda |Z_i|_1$$

where $x_i$ is an input vector, $\lambda$ is the regularization parameter, $D$ is the dictionary, and $Z_i$ is the sparse feature vector. If $x$ has $p$ dimensions, $Z$ has $m$ dimensions ($m > p$), then $D$ forms a $m \times p$ matrix (or a 2-D array). To quickly reach a stable sparse representation for $x_i$, state-of-the-art algorithms apply iterative, parallel, or stochastic methods for the two most computationally intensive tasks: updating the feature vector $Z$ and updating the dictionary $D$.

Note that we assume non-negative sparse coding Hoyer (2002), where both the dictionary and the sparse codes are assumed to be all non-negative. This is justified both because of its simpler representation and the fact that the dictionary values are encoded directly in hardware physical properties (e.g. conductance of RRAM devices), and the input data (e.g. image) is usually non-negative.

In this paper, we focus on the Iterative Shrinking-Thresholding Algorithm (ISTA) Daubechies *et al.* (2004) to update $Z$ due to its inherent algorithmic parallelism, and the Stochastic Gradient Descent (SGD) Bottou and Bousquet (2008) to update $D$ exploiting stochasticity for greater efficiency. The steps of updating the sparse code and the dictionary using the algorithms are described below. The image patch size used in this paper is 10×10, which will densely sample the entire input image.

1. *Randomly initialize the dictionary*

2. *Get an image patch* $x_t$

3. ***Update Z*** *via ISTA*:

$$r_{t+1} \leftarrow D \cdot z_t^k - x_t \ ,$$

$$b \leftarrow z_t^k - D^T r / L \ ,$$

$$z_{t,j}^{k+1} \leftarrow \max\{0, b_j - \lambda / L\}$$

   Where $b_j$ and $z_{t,j}^{k+1}$ denote the $j$-th element of $b$ and $z_t^{k+1}$, and $/L$ is the soft thresholding function, and $r_t \triangleq D_t \cdot Z_t - X_t$ is the residual error of data presentation ($r$).

4. ***Update D*** *via SGD*:

$$\hat{D} \leftarrow D - \eta_t r z_t^T$$

$$D \leftarrow B(D)$$

Where $\eta_t$ is the learning rate and $B$ denotes the projection operator onto the unit $l_\infty$ ball i.e, if $D_{ij} < 0,\ B(D_{ij}) = 0$. Hence if $D_{ij} > 1,\ D_{ij} = 1$. This way $D$ is bounded and the sparse code $Z$ cannot be arbitrarily large.

These learning algorithms are typically implemented in software, and run on a general-purpose CPU/GPU. Limited by the sequential architecture of today's microprocessors, they suffer from long computing times, especially in dealing with a large $D$ matrix. Thus, it is desirable to have a special hardware that accelerates the learning process beyond such limitations.

Noticing the similarity of plastic synapses in biology and the learning dictionary in sparse coding, we intend to adopt suitable biological principles such as local programming and spike-based communication while we implement learning algorithms into an efficient hardware platform that accelerates learning.

Bio-inspired sparse coding has been recently demonstrated on CMOS ASIC hardware Kim *et al.* (2014a,b) that implements the SAILnet algorithm Zylberberg *et al.* (2011). SAILnet based sparse coding is efficiently implemented in Kim *et al.* (2014a,b) using memory partition (between learning and inference), approximate learning, and a systolic ring architecture. Local computing and updates are common features, but the proposed work could be differentiated from Kim *et al.* (2014a,b) with respect to algorithm, device, and the supporting circuitry. To minimize the objective function in dictionary learning Tosic and Frossard (2011), we used algorithms based on convex relaxation methods such as least absolute shrinkage and selection operator (LASSO), which can be shown to converge. Among them, we identified ISTA and SGD algorithms as those that would map efficiently on RRAM based hardware. Using the analog weights of RRAM devices and custom peripheral circuitries, we propose to parallelize the Read (matrix-vector multiplication) and Write (weight update) operations in a parallel manner, speeding up the heavy iterative computations in sparse

coding.

The resistive crosspoint array structure, shown in Fig. 2.1(b), was recently proposed as a promising solution for learning in hardware neural networks Afifi *et al.* (2010); Rajendran *et al.* (2013). The iterative solution to the sparse coding problem can be realized by mapping the matrix $D$ onto the resistive array, and learning takes place through the update step. The input vector $X$ (or $r$) is associated with one side of the array and the sparse feature vector $Z$ with the other side. In this way, the crosspoint mimics the structural map of a neural system. At each cross point, the conductance ($G$) of a memory cell represents the synapse weight. The memory technology of choice is resistive random access memory (RRAM), due to its non-volatility, integration density, and low power consumption Jo *et al.* (2010). The inset of Fig. 2.1(b) illustrates its structure.

Using the physical properties of RRAM devices, we have implemented a variation of STDP where the timing between spikes is not considered but the synaptic weights or dictionary values are updated based on the number of the spikes (spiking rate) in a given timing window. With this learning rule that can be described as time-dependent synaptic plasticity, $G$ of a RRAM cell is increased (or decreased) by positive (or negative) voltage pulses or spikes occurred in a certain timing window. The two core functions we propose to implement using the resistive crosspoint array include:

1. ***Read for Matrix-Vector Multiplication***: When a voltage is input from $Z(V_{Z,j})$, the output current at $x_i$ is $I_{X,i} = \sum G_{ij} \cdot V_{Z,j}$. If $G$ encodes $D$, then a Read corresponds to sensing the current which encodes $D \cdot Z$, which is computed in parallel.

2. ***Write to Update D***: The conductance of the entire array is updated in parallel. Previous approaches involve sequential operations (row-by-row, column-

by-column, or even bit-by-bit) to update $G$ of the RRAM cell.

The large dimension of $D$ (i.e., large fan-in and fan-out to each $X$ and $Z$ node) could pose unique challenges to the periphery circuit design: for Read, the receiver needs to convert a tremendously wide range of output current $I_i$ (>100× difference) to a digital data at high precision; for Write, it is preferred to program all cells in parallel for high-speed computation, with local data only from pre-synaptic and post-synaptic nodes, as observed in a biophysical synapse.

In this paper, we present effective solutions to these challenges by integrating CMOS circuitries at the periphery of a resistive crosspoint array. Particularly, a current-to-digital converter ($D \cdot Z$) is designed for the Read operation, and a spike generator ($r$) and a timing window generator ($Z$) is designed for the Write operation. The operation of these peripheral circuits together with RRAM devices has been verified by array-level simulation.

The remainder of the paper is organized as follows. We present the resistive memory device and the underlying physical device properties in Section 2.2. Section 2.3 describes the parallel architecture and principles of Read and Write circuitries. Section 2.4 presents experimental results from a 65nm CMOS design, and learning demonstration is shown in Section 2.5. The chapter is concluded in Section 2.5.

## 2.2 RRAM Device

A RRAM (also referred to as memristor) is a two-terminal electronic device whose conductance can be modulated by voltage stimulus applied on the two terminals. Since the conductance remains unchanged until it is modulated again, it can be used as a memory device that stores continuous conductance values. The multi-level RRAM is analogous to a biological neuron synapse as it is massively parallel, three-dimensionally organized, extremely compact, power efficient and combines storage

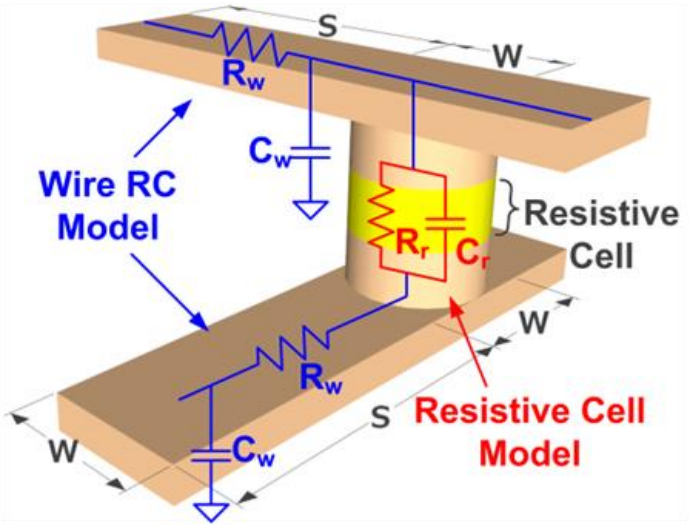



**Figure 2.2:** Sub-circuit module of a single resistive cell ($S$ : cell spacing; $W$ : wire width). The cell capacitor ($C_r$) is in parallel with the cell resistor ($R_r$). The wire resistors ($R_w$) and capacitors ($C_w$) for top and bottom interconnect are considered. Sub-circuit is duplicated in simulations for the array.

and computation Kuzum *et al.* (2013). In a cross-point array, the RRAM cells is formed at each cross-point between the rows and columns, which emulates the neural networks in a 2-D array fashion where the synapses connects pre-synaptic and post-synaptic neurons. A RRAM model emphasizing the parasitics for a single device is shown in Fig. 2.2. Using multi-level RRAM devices for synaptic devices have been demonstrated with various device materials stacks including Ag:a-Si Jo *et al.* (2010), HfOx/TiOx Yu *et al.* (2013) PCMO Park *et al.* (2013), and TaOx/TiOx Wang *et al.* (2014).

RRAM has the advantage of being a programmable memory which can also be fabricated at high density (RRAM cell size could be $4F^2$, where $F$ is the feature size of a technology node). The programming of the device is performed by keeping a programming or write voltage $V_w$ (in our case, set to be $Vdd$) across the device for a certain amount of time. Ideally the change in conductance of a single device is linear, which allows for the conductance to return to the initial value if the same

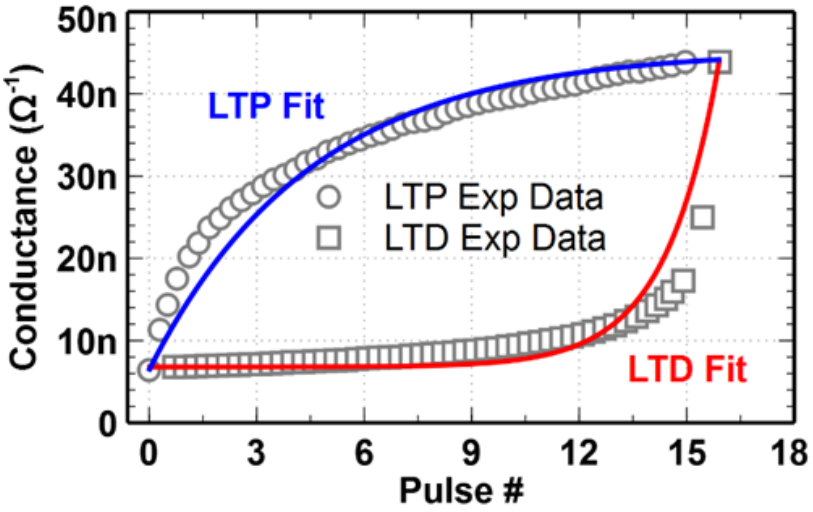



**Figure 2.3:** Experimental data Park *et al.* (2013) of conductance modulation in RRAM based synapse in linear scale is shown along with the best fit graph of the raw data.

number of positive and negative pulses are applied to the RRAM device. Since such an ideal device does not exist yet, we use a model obtained from experimental data based on PCMO Park *et al.* (2013). Using the model, Fig. 2.3 shows how the RRAM conductance changes with linear increase in the number of pulses. Long-term potentiation (LTP) is the steady increase in the conductance of the RRAM, which is the result of a constant potential $V_w$ applied across the device. On the other hand, long-term depression (LTD) occurs when the potential difference across the device is reversed, resulting in decrease in RRAM conductance. The equations governing this change were extracted and used for the simulations and are given below in Eq. (2.1) and Eq. (2.2). $G_{min}$ and $G_{max}$ are the minimum and maximum conductance possible for a single device. Due to the strong non-linearity in the voltage dependence on the conductance change in RRAM devices, we can safely assume that the RRAM conductance would not change if half of the write voltage is applied across the two terminals, which is a property that we will leverage to design the parallel write scheme

in Section 2.3.3.

$$G_{LTP} = B\left(1 - e^{(-\frac{X}{A})}\right) + G_{min} \tag{2.1}$$

$$G_{LTD} = B\left(1 - e^{(\frac{X - X_{max}}{A})}\right) + G_{max} \tag{2.2}$$

The read operation attempts to measure the stored conductance value by applying a read voltage (that is small enough to ensure the programmed conductance does not change) across the two terminals of the RRAM device and sense the current value. In a crosspoint array, we read out the weighted sum of read voltage and the RRAM conductance through all the devices in one column, and these read operations of multiple columns are performed in parallel. Further details are presented in Section 2.3.3.

We implemented the RRAM conductance dynamics in Eq. (2.1) and (2.2) into a Verilog-A model, which govern the behavior shown in Fig. 2.3. Besides the RRAM cell, this Verilog-A model also includes the interconnect wire resistance and parasitic capacitance as was illustrated in Fig. 2.2. These components will be used together to analyze the IR drop issue across the row/column and the sneak path concern Liang *et al.* (2013) in the crosspoint array later in Section 2.4. The robustness of the hardware operation against spatial and temporal variations of RRAM devices are important challenges in the array design. Being investigated by the authors in a separate publication, RRAM device variation will be out of the scope of this chapter.

## 2.3 Crosspoint Array Architecture and Design

### *2.3.1 Overall Architecture of PARCA*

Fig. 2.4 illustrates the proposed Parallel Architecture with Resistive Crosspoint Array (PARCA). The $D$ array connects $Z$ on one side and $r$ on the other side. Devices connected together in one row or column are separated by a wire resistance $R_w$, as

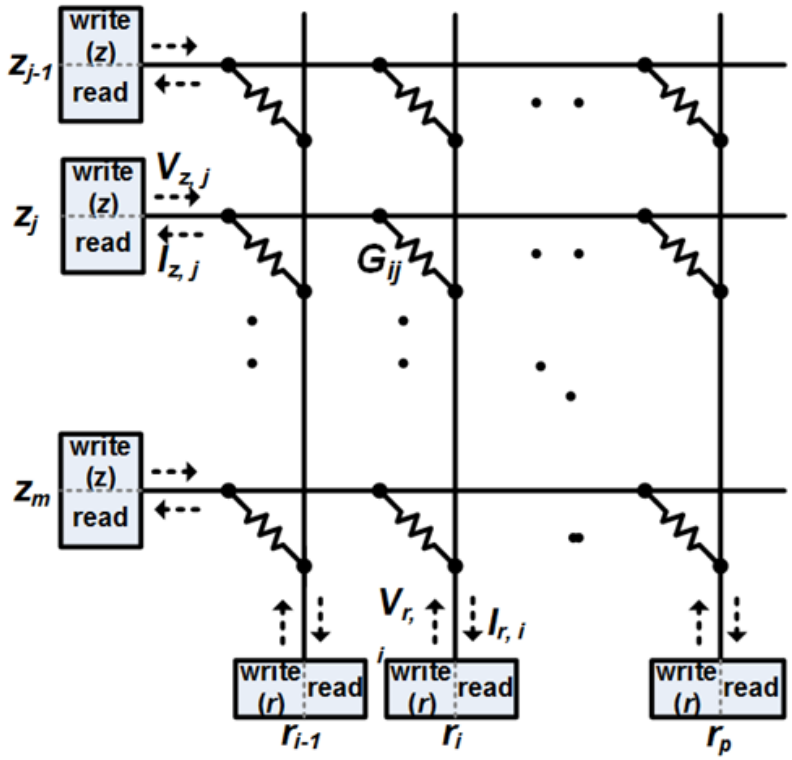



**Figure 2.4:** PARCA architecture with peripheral Read and Write modules. $Z$ and $X$ (or $r$) nodes have the same Read (Section 2.3.2), but different Write circuits (Section 2.3.3). All RRAM cells perform Read or Write operation in parallel.

was shown in Fig. 2.2. We will investigate the ramifications of IR drop in Section 2.4. The two key operations that we intend to fully parallelize are: $D{\cdot}Z$ and $D$ update.

- $D{\cdot}Z$ (or $D^T{\cdot}r$): Parallel Read of the RRAM array. For each non-zero bit of $Z$, a small read voltage is applied simultaneously. The read voltage $V_Z$ is multiplied with $G$ at each crosspoint, and the weighted sum results in the output current at each $r$ node. The read circuits described in Section 2.3.2 convert this current into a binary number. Compared to conventional memory arrays that require reading row-by-row, our approach reads the entire RRAM array in parallel. If $Z$ has the data precision of $m$ bits, the read operation will be performed $m$ times (once for each bit) and the current is read out for each iteration. As we go through $m$ read operations, the final $D \cdot Z$ value is obtained from shifting

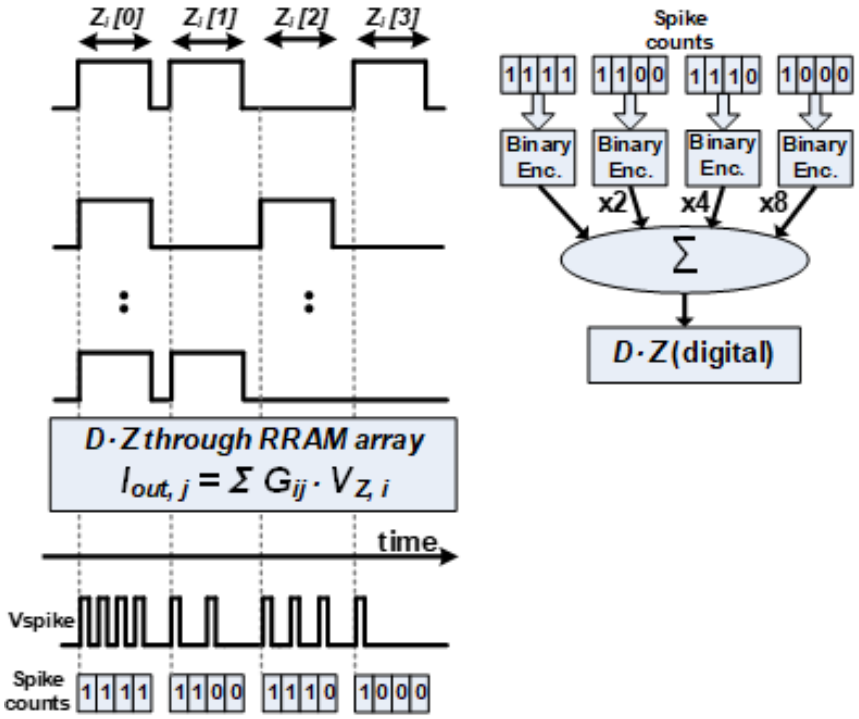



**Figure 2.5:** Procedure to obtain the final $D \cdot Z$ value from read out values of single bits. The final operation involves encoding the spike counts to binary values and summing them.

and adding the values acquired from each iteration, similar to the multiply and accumulate (MAC) computation. This process is illustrated in Fig. 2.5, where we show how the current read out for each bit of $Z$ is combined to get the final $D \cdot Z$ value for a single column or row. A similar Read operation in the transpose direction computes $D^T \cdot r$.

Compared to conventional memory arrays that are constrained by row-by-row read operation, PARCA is not bounded by such serial operation, and allows for the read operation to be performed in parallel for the entire array for each $Z$ bit applied simultaneously at all the rows. Note that one bit is applied in one cycle at each row in conventional arrays as well. For each $Z$ bit applied on all the rows, this one-shot read operation does not depend on the RRAM array size, which provides large amount of acceleration. As shown in the Fig. 5, the output from the read circuit is a stream

**Table 2.1:** PARCA Operations for Key Sparse Coding Tasks.

| **Task** | **PARCA Method** |
|---|---|
| $D \cdot Z$ | $I_{r,i} = \sum_i G_{ij} \cdot V_{Z,j}$ |
| $D^T \cdot r$ | $I_{Z,j} = \sum_j G_{ij} \cdot V_{r,ij}$ |
| $D$ update | $\Delta G_{ij} = \eta \cdot r \cdot Z$ ($\eta$ is the learning rate Bottou and Bousquet (2008)) |
| Read | Input: small $V$ pulse; Output: $I$ to digital |
| Write | Input: large $V_r$ and $V_Z$ pulses, with proper timing between them |

of 1's which represent the number of spikes that have occurred, and a simple 8-to-3 bit thermometer code to binary encoder is employed. The need for the thermometer code to binary encoder is to convert the 8-bit thermometer code output of the read circuit to a 3-bit binary code, subsequently to acquire the final output discernable by other circuits to process. For each $Z$ bit, the output of the read circuit is a 3-bit digital representation of the analog weighted-sum current. After going through all 4 bits of $Z$, we obtain the final 6-bit output value, which will be used in subsequent computations.

- $D$ update: Parallel Write of the RRAM array. The write operation is performed by changing the stored dictionary value in the crosspoint array. In SGD algorithm, the change of $D$ is proportional to $r \cdot Z$ Bottou and Bousquet (2008). By properly generating voltages at local $r_i$ and $Z_j$ nodes, the current conductance $G_{ij}$ of a RRAM cell is changed by an amount proportional to $r_i \cdot Z_j$. Thus, all RRAM cells are modified in parallel, achieving considerable speedup compared to previous approaches that require read-modify-write operations. The proposed write circuitry is described in Section 2.3.3. Table 2.1 summarizes the key operations handled by PARCA.

### 2.3.2 Read: Integrate and Fire

The proposed Read circuit is essentially a current-to-digital converter, where it senses the output current at each $r_i$ (or $Z_j$) node for $D \cdot Z$ (or $D^T \cdot r$), and converts to digital values Kadetotad *et al.* (2014). In principle, this output response is similar to that of a biological neuron model, namely Integrate-and-Fire (IF) Abbott (1999); Mead and Ismail (2012). Starting from a reset voltage, the output current is integrated on the finite capacitance of each RRAM column; when the voltage charges up above a certain threshold, the output switches and the capacitance is discharged back to the reset voltage. The read property of a RRAM cell further poses a constraint that the reset voltage and the threshold voltage should be very close to each other; otherwise the output current does not represent the correct weighted sum Yu *et al.* (2013)Wong *et al.* (2012). In our 65nm design, the reset voltage and the threshold voltage are 500 mV and 530 mV, respectively. To meet this constraint, an asynchronous comparator with high sensitivity to small changes in voltages was required, and we employed a Schmitt trigger Wang (1991)Katyal *et al.* (2008) to generate output spikes proportional to the input current. We explored both single-ended adaptive design and differential design to investigate trade-offs that include power, variation, and performance.

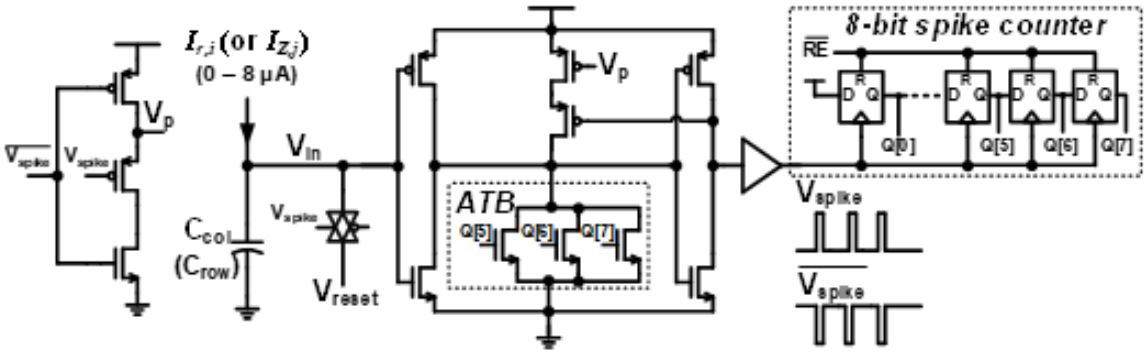



**Figure 2.6:** Schematic of the single-ended adaptive Read circuit is shown. Based on the IF neuron model, it converts an analog input current $I_{r,i}$ into a digital number.

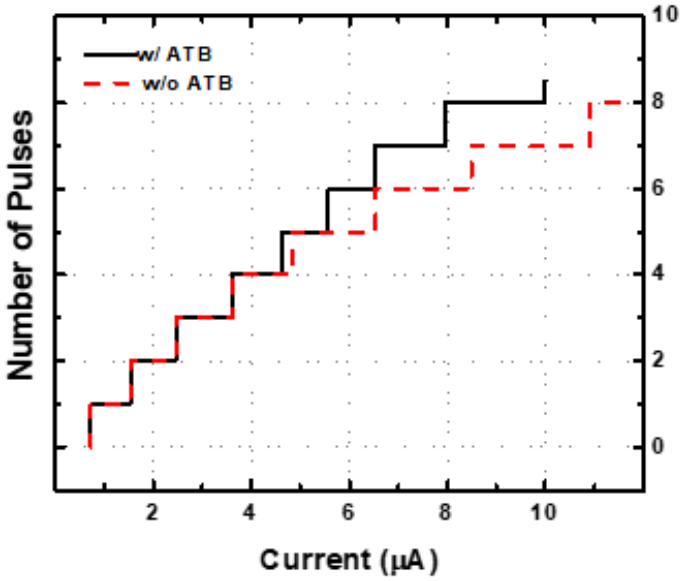



**Figure 2.7:** Effect of ATB on single-ended adaptive Schmitt trigger based read circuit.

- *Single-ended Adaptive Schmitt Trigger*: For $D \cdot Z$, we measure the integrated current at each $r_i$ node by counting the number of times ($n_i$) the voltage at the integration node crosses the set threshold within a read timing window ($T_R$). As the charge accumulates over time on a finite capacitance, the time it takes for the integration voltage to exceed the threshold is inversely proportional to the current ($I \cdot t$ = constant). Since $n_i$ $1/t$, the current will be proportional to the number of spikes that occurred during a fixed timing window. Fig. 2.6 shows the Read circuit where the capacitance used to integrate the current is the parasitic capacitance of the RRAM column or row. The transmission gate (TG) discharges the capacitance while the adaptive threshold block (ATB) strengthens the pull down network to vary the threshold below 530 mV only when the incoming current is high. The output of the Schmitt trigger is buffered and drives the clock input of a 8-bit shift register to store $n_i$. Fig. 2.7 shows the how ATB affects linearization in the current-to-digital conversion operation of the read circuit. This is achieved by trading off 10% power consumption, due

to the current flowing through the intermediate branch of the Schmitt trigger.

- *Differential Schmitt trigger*: The differential Schmitt trigger that uses the architecture described in Yuan (2010) is shown in Fig. 2.8. The differential pair allows for precise calibration of the voltage at which the reset pulse should be activated. The hysteresis loop is created by the cross-linked tail to the differential pair. By properly sizing the transistors in this configuration, the thresholding operation was enabled. Also the read window is decreased from 4.5 ns in the single-ended design to 4 ns in the differential design by enhanced latency, while trading off additional power consumption of 7.4% compared to the single-ended design, Unlike the $V_p$ voltage that changed with output spikes in the single-ended design, the differential design operates with a constant $V_p$ voltage of 500 mV . Furthermore, the improved linearity eliminated the need of the ATB block, as the circuit compares $V_{in}$ to the reference voltage, the ATB was not required to force a switch at the granularity of 1A which reduced area.

The effect of process variations during manufacturing on the correctness of the current-to-digital conversions will be discussed further in Section 2.4.4. By careful symmetric layout and the help of common noise rejection of differential voltage comparison, the differential Schmitt trigger based read circuit could be far less susceptible

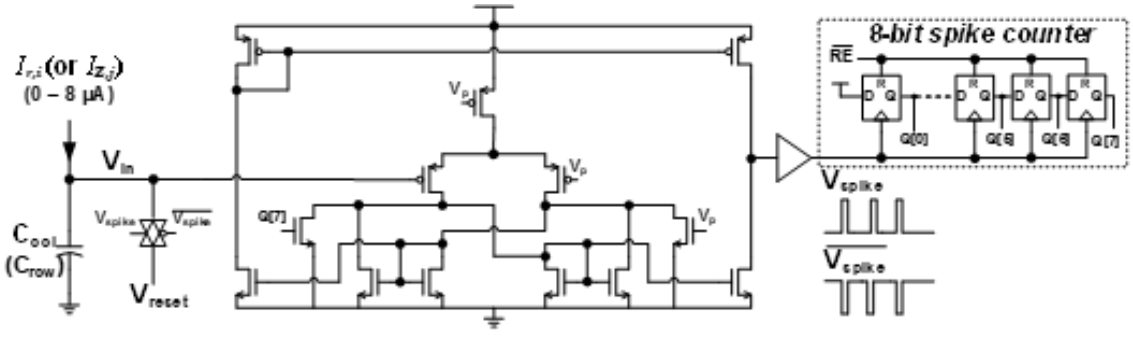



**Figure 2.8:** Schematics of the differential Read circuit. Latency and sensitivity to variation is improved compared to the single-ended design.

to variations compared to the singled-ended design.

- *Read for Negative $r$*: In the sparse coding algorithm, the value of $r$ could be positive or negative when $D^T \cdot r$ is computed. Since there is no direct communication between the column-side $r$ driver and Read circuits other than that through the crosspoint array, we also designed a read circuit configured for negative $r$ values, which gets activated only when the input voltage $V_{in}$ drops below 500 mV.

For the singled-ended adaptive read circuit, the input NMOS transistor is sized larger and the polarity of $V_{spike}$ in the transmission gate connection is swapped for read operations for negative $r$ values,. For the differential Schmitt trigger, the inputs for the differential input pairs are simply swapped to enable the negative read operation.

### *2.3.3 Write: Timing based Local Programming*

To change the conductance of an RRAM cell, the voltage across the cell should be $Vdd$, and $Vdd/2$ only induces negligible change on $G$ due to its strong dependence on the voltage as discussed earlier. Inspired by time-dependent synaptic plasticity, $G$ is programmed by the overlap time between local $r$ and $Z$ signals [26]: The write circuit for $Z$ generates a pulse with a duty cycle proportional to $Z$, while a spike train is generated at $r$ with the firing rate proportional to $r$ and the pulse width is fixed at 1ns. Wherever the pulse at $r$ is overlapped with $Z$, it creates $|V_Z - V_r| = Vdd$. Therefore, the total programming time equals to the overlap between $Z$ and $r$, i.e. $r \cdot Z$. Since $Z$ is always positive while $r$ can be positive or negative, we divide the write period into a positive/negative period for $r > 0$ / $r < 0$.

1. *Write Circuits for $Z$*: In the positive period, $Z$ is 0 for a certain time proportional to $Z$; then it switches to $Vdd/2$. The overlap time between $Z = 0$ and

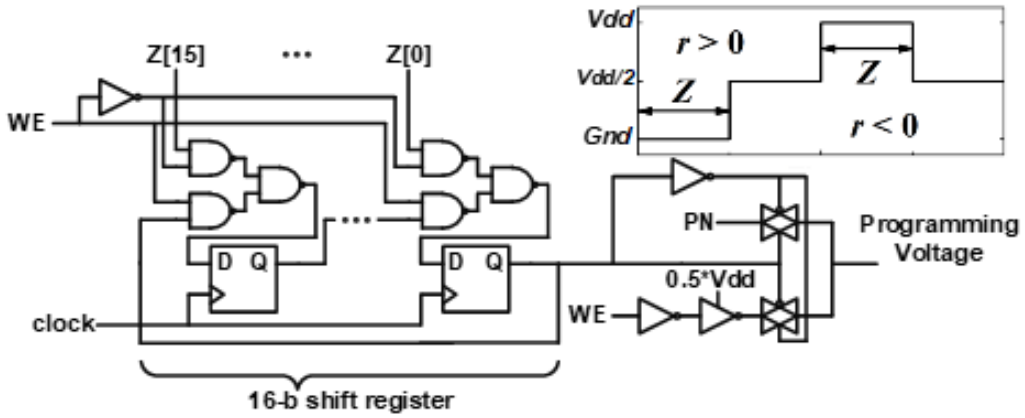



**Figure 2.9:** Write circuit for $Z$, which generates a timing window that is proportional to $Z$.

$r = Vdd$ tunes the RRAM conductance. A similar scenario is designed for the negative period, with a symmetric polarity.

Fig. 2.9 shows the digital design to generate such a pulse pattern. A 16-bit shift register converts the parallel input $Z[15:0]$ into a sequential output. The time when the output is 1 is proportional to the value of $Z$. The output of the shift register is connected back to the first stage of itself in order to recycle the data $Z$. With 32 clock cycles allocated for one write period, the $Z$ write circuit generates two identical pulses with the duty cycle proportional to the value of $Z$. These two identical pulses are connected to multiplexors to generate different programming voltages for the positive period and the negative period.

1. *Write Circuits for* $r$: The train of pulses generated at $r$ has its pulse number proportional to the value of $r$, where each pulse has a fixed width for fixed RRAM programming period. The pulses are evenly distributed across the write period in order to minimize the quantization error.

Fig. 2.10 presents the design for generating the $r$ signal. It consists of various delay elements forming a configurable ring oscillator (RO) with the polarity control

by the sign-bit of $r$. The number of pulses during the write period (i.e., the firing rate) is varied by adding switches into the oscillation loop, which determine the total gate delay in the ring oscillator. The control signal of the switches is generated from the $r$ value, ensuring that only one switch is on for a particular value of $r$. When $r = 0$, no change in the RRAM conductance is allowed. In total, 15 buffer stages (d1-d15) are implemented with different delay values, such that the number of pulses generated in each write cycle is proportional to the $r$ value. From each rising edge of the RO output, the pulse generator generates a pulse with fixed 1ns width. This ensures that the total programming time is proportional to the pulse number for our RRAM technology. The sign-bit of $r$ and the write phase (PN) finally select the output signal among $Vdd$, 0, the pulse generator output or the inversion of it.

The proposed $r$ circuit can provide 16 different pulses during one write cycle. To ensure that the required 64 levels of granularity of the weight update is upheld, the aforementioned write operation is repeated four times, with corresponding $Z$ and $r$ values.

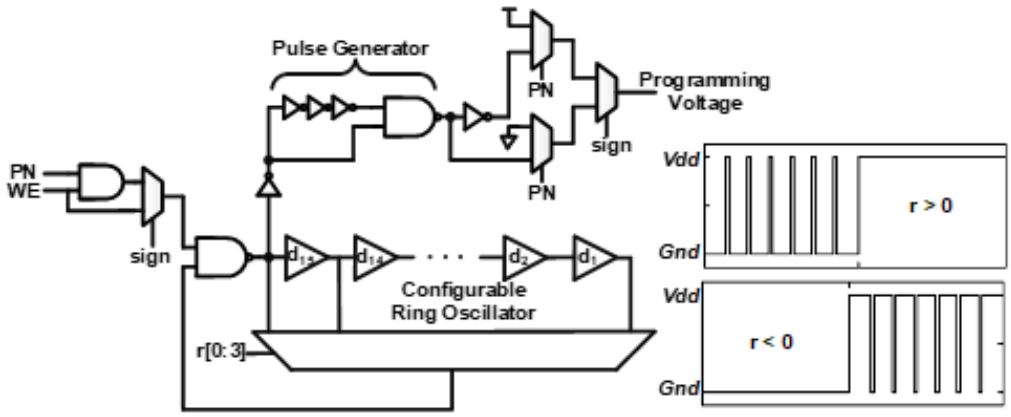



**Figure 2.10:** Write circuit for $r$, which generates a series of evenly-spaced spikes whose firing rate is proportional to $r$.

## 2.4 Simulation Results

The read and write circuitries proposed in Section 2.3 are designed in TSMC 65nm CMOS technology. These peripheral circuits are simulated together with our RRAM Verilog-A model described in Section 2.2. In this section, we show the corresponding read and write simulation results with a single RRAM device as well as a sizable RRAM array (100×100). The array size of 100×100 is chosen to fit with 10x10 image patches we use, where one pixel in the image patch corresponds to one column in our RRAM array. When capturing local features of the image such as edges, the size of the image patch would provide trade-offs among the array size, the complexity of a single-stage operation, and the number of such operations that is required in series. For the single-stage operation with a specific image patch size (10x10 in our design), the speedup in the proposed PARCA is independent of the matrix size, since PARCA is not constrained by conventional row-by-row operations, and read and write operations could be performed for the entire array (matrix).

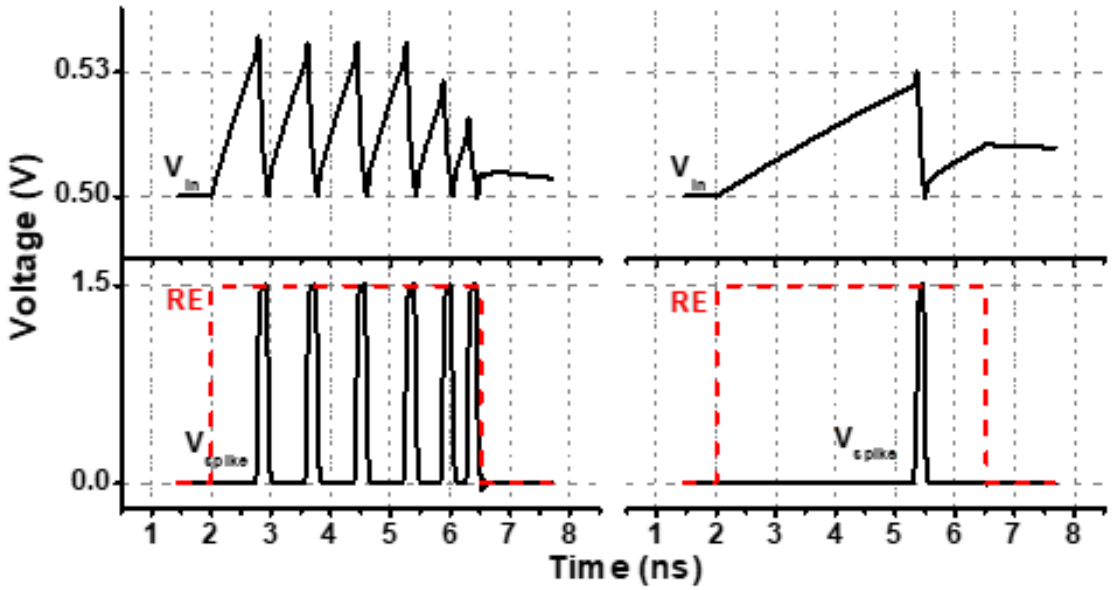



**Figure 2.11:** The operation of the single-ended read circuit for two input currents: (left) $I_r = 5.75\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1.

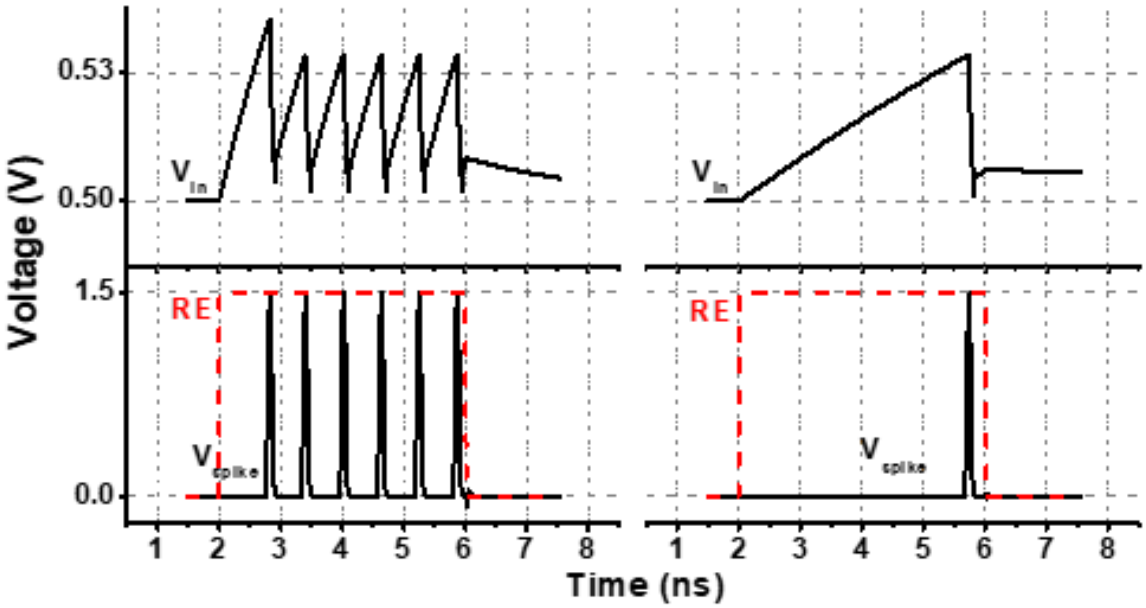



**Figure 2.12:** The operation of the differential read circuit for two input currents: (left) $I_r = 5.75\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1.

### *2.4.1 Simulations with single RRAM device*

1. *Read*: Fig. 2.11 and Fig. 2.12 demonstrate the proper operation of the read circuit for positive values of $r$, for the single-ended adaptive design and the differential design, respectively, with two values of input current. The RRAM current integrates at the input node ($V_{in}$), increasing the voltage until it reaches the threshold of the Schmitt trigger. The circuit then initiates reset to discharge the capacitance. This integrate-reset process continues while Read Enable (RE) is high. The number of reset pulses ($n_i$) present in this timing window (4.5 ns and 4 ns in our single-ended and differential designs, respectively) is recorded by enabling the shift register for each reset pulse. This functionality is shown for both read circuits.

In Fig. 2.13 and Fig. 2.14, the read operations with negative $r$ values are demonstrated. At each iteration, the input voltage decreases until it crosses the threshold,

at which point it is reset to 500 mV. The read circuit for negative $r$ has the same quantization error as its positive counterpart in both the single-ended adaptive as well as the differential read circuit.

1. *Write*: Fig. 2.15 shows the timing diagram of the parallel programming system with programming time of 84 ns. When the write enable (WE) signal turns on, both $Z$ and $r$ write circuitries start generating the pulses based on the values of $Z$ and $r$, thus changing the value of $D$ during the overlap time. Fig. 2.15 demonstrates the write operation when $r$ is positive, where the programming occurs in the positive period and the value of $D$ decreases; when r is negative, the programming happens in the negative period and the value of $D$ increases.

### *2.4.2 Simulations with 100×100 RRAM array*

To analyze the two core functions of PARCA in a sizable array, we integrated 100 rows and 100 columns of RRAM device models as described in Fig. 4(a). We verified

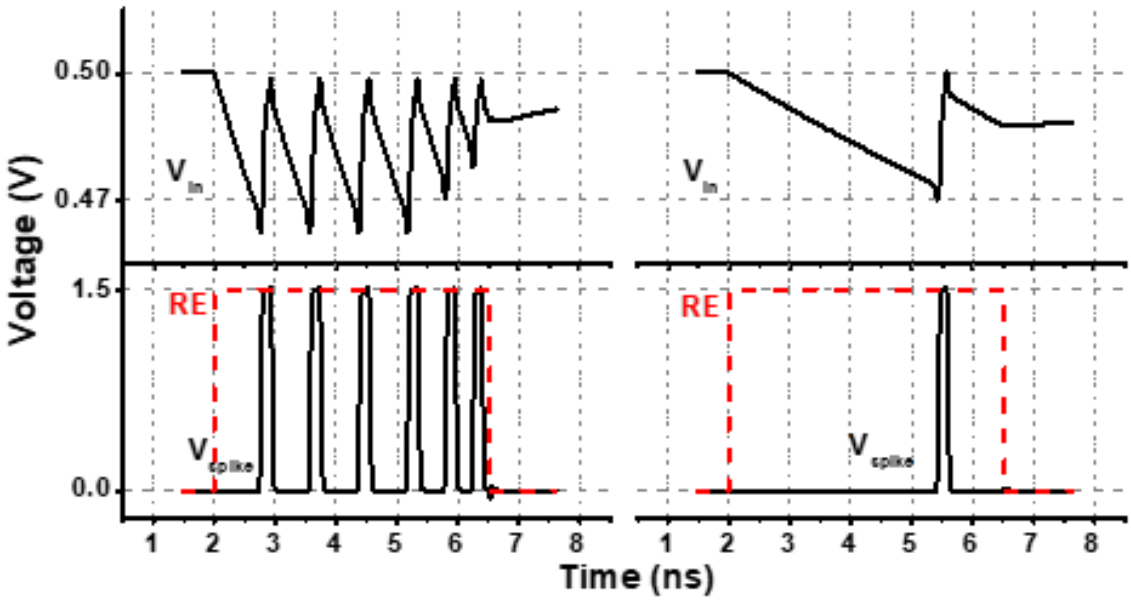



**Figure 2.13:** The operation of the single-ended read circuit for two input currents: (left) $I_r = 6\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1.

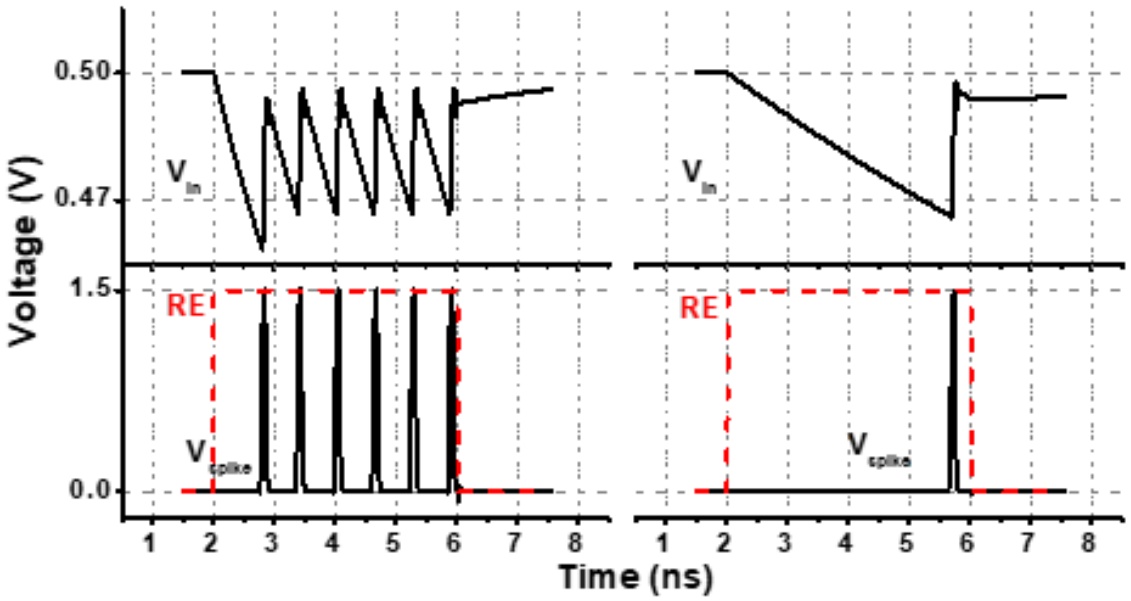



**Figure 2.14:** The operation of the differential read circuit for two input currents: (left) $I_r = 5.75\mu A$; and (right) $I_r = 1\mu A$; the corresponding $n_i$ is 6 and 1.

the Read and Write operation and compared this to the single device simulation. The read and write circuits are designed in 65nm CMOS technology, and simulated together with the RRAM model is calibrated with the data from Lee *et al.* (2010). We formed an array with the Verilog-A models of the RRAM devices together with wire parasitics. Fig. 2.16 and Fig. 2.17 show the simulation results of such array-level

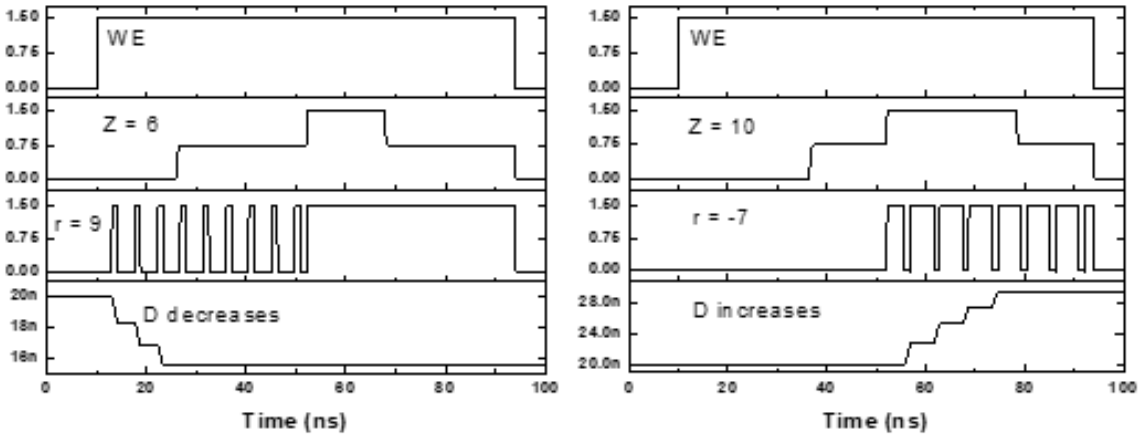



**Figure 2.15:** The overlap in time between $Z$ and $r$ pulses modulates $D$.

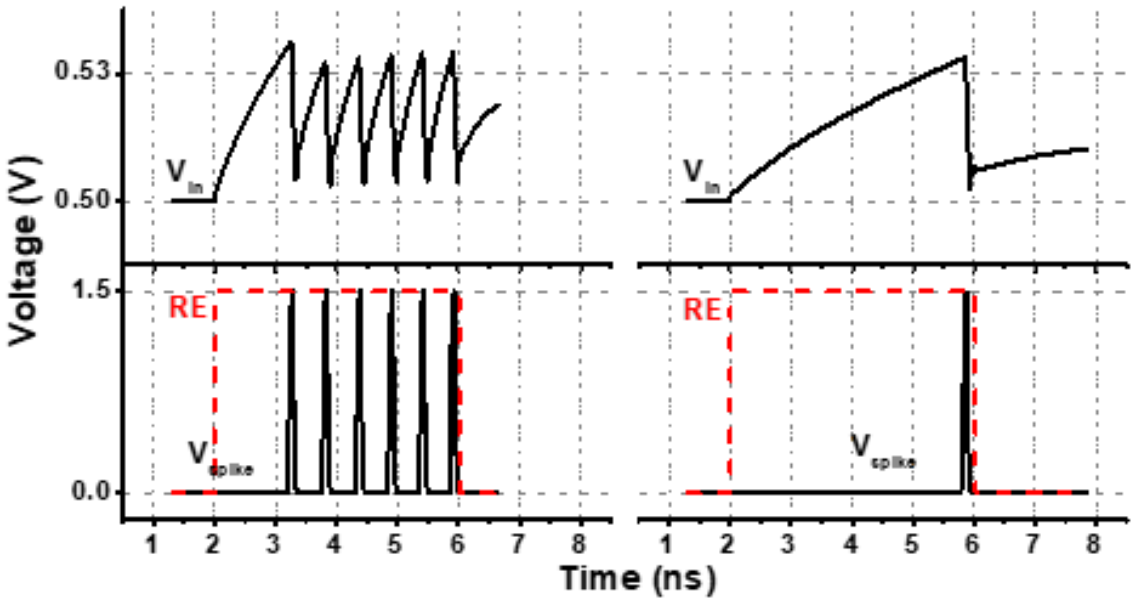



**Figure 2.16:** 100×100 array simulation of differential read circuits for two input currents: (left) $I_r = 5.55\mu A$; and (right) $I_r = 1.25\mu A$; the corresponding $n_i$ is 6 and 1.

read and write operations.

1. *Read*: Fig. 2.16 show the operation of a read circuit for an RRAM array of 100×100 dimension including all measured parasitics of the device. We can observe the small effect of sneak paths in the nonlinear integration of $V_{in}$. This causes a rightward shift of the staircase waveform in Fig. 18(a), but the design parameters in the proposed read circuits could be calibrated the current-to-digital conversion for a specific array size. Apart from the sneak paths, adjacent columns integrating and resetting at different rates may effect other columns through capacitive coupling but this effect has not been noticable in our simulations.

2. *Write*: Regarding the array level simulation of the write operation, the primary concern is the IR drop caused due to the wire resistance $R_w$ which in turn results in different $V_w$ for devices in same column or row. This effect is most noticable

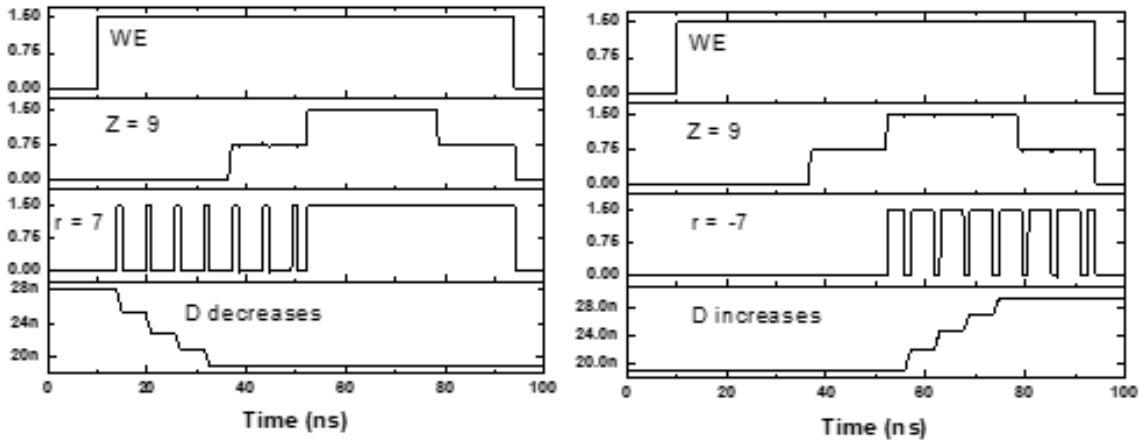



**Figure 2.17:** 100×100 array simulation for write operation by overlapping $Z$ and $r$ pulses.

for the RRAM devices furthest from the periphery where the write circuits for $Z$ and $r$ are present. Also the combined wire capacitance $C_w$, which is an integral part of the read circuit causes slow rise and fall of the write circuits. To avoid this problem, the write circuits are equipped with buffers which can drive the capacitive load accordingly presented by the RRAM array. The write operation for the entire array is demonstrated in Fig. 2.17, where the conductance of the corresdponding RRAM device in the array is shown to change through the course of the simulation.

### *2.4.3 Quantization Error*

Fig. 2.18 shows that quantization error is present in both read and write circuits, which must be kept small for the correct functionality of the algorithm. As shown in Fig. 2.18(a), the number of reset pulses linearly increases with incoming RRAM current at ∼1A granularity. Non-linearity exists at high $G$ values, which is due to the finite discharge time of the capacitance and the voltage overshoot above threshold due to latency. The non-linearity further limits the lower bound of the read time window, forcing a longer read time. In the single-ended adaptive design, the proposed ATB

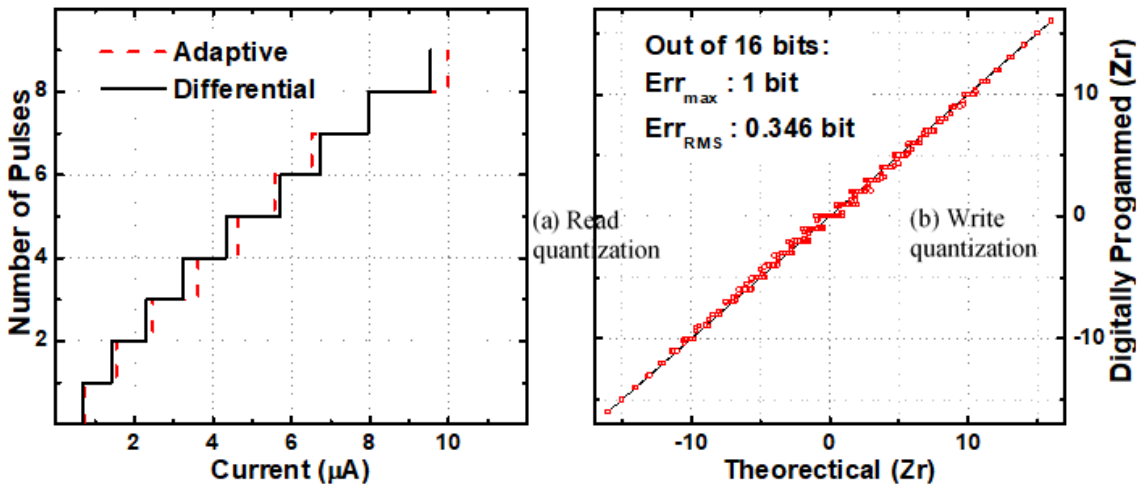



**Figure 2.18:** Quantization of read and write circuits are shown. (a) Number of pulses and RRAM current show a close-to-linear relationship. (b) Digitally programmed pulse width closely follows the mathematical multiplication.

unit is only enabled when the conductance is high, to ensure high linearity between $G$ and $n_i$.For the single-ended adapative design, the introduction of the ATB while the write quantization error is small and managable.

The method of using overlap time of $Z$ and r pulses with a certain granularity to calculate $r \cdot Z$ introduces quantization error. To analyze this, we performed simulations for all $Z$ and $r$ values. Fig. 2.18(b) compares the simulated results to an ideal multiplication. The digital programming closely follows the theoretical value, while producing the maximum error of 1 bit (out of 16 bits) when both $Z$ and $r$ are small.

### *2.4.4 Sneak Path and interconnect in RRAM*

As described in Lee *et al.* (2010); Xu *et al.* (2011); Zhou *et al.* (2014), sneak paths can occur when multiple parallely connected RRAM devices have different potentials across them. This problem is exacerbated in our case as many rows become inactive while a small portion (∼20%) of them are active due to the sparse nature of the algorithm. As illustrated in Fig. 2.19, there exists a number of alternate paths for

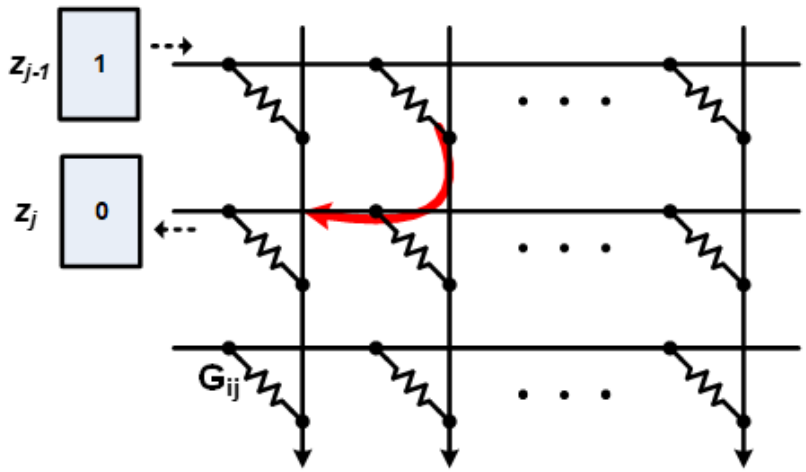



**Figure 2.19:** Sneak paths could cause current to flow through inactive nodes during the read or write operation.

the current, which causes non-linear integration of current during the read operation. We conducted experiments on two methods to control to fix the sneak path issue Xu *et al.* (2011). First, we fixed the inactive rows to 500mV (steady-state column voltage potential). Secondly, after precharing the inactive rows to 500mV, we then let the inactive nodes float, which causes the inactive row or column nodes to increase in voltage together with the adjacent active nodes, reducing the potential difference between the active and inactive nodes. Particularly, for a $D \cdot Z$ operation, the rows with non-zero Z values are driven with a read voltage, whereas the rows with zero-Z values floats by tri-stating the row driver. For a $D \cdot Z$ read simulation, we found out that the floating node approach resulted in linear read current quantization (Fig. 2.18(a)), while the fixed bias approach increased the charge accumulation time by up to 25%, significantly distorting the linearity.

The write operation could also induce sneak path currents, when the $Z$ or $r$ driver current sneaks into different node paths causing erroneous outputs. However, our array-level write simulation shows that the amount of current sneaking into other nodes during the write operation is 41.8$n$A in the worst case and averages at 15.87$n$A,

which is insignificant and does not affect the potential difference generated by the write circuits. The IR voltage drop caused by the wire resistance of the RRAM array could hamper the accuracy of the write operation moderately. Using wire width of 200$n$m for M6-M7 wires in 65nm, the resistance ($R_W$) in Fig. 2.2 results in 633.8$m$W. The write simulation results showed that the voltage drop across the entire array due to the wires is 2$m$V, which does not affect the functionality of the write operation.

### *2.4.5 Variability and Mismatch*

Monte Carlo simulations were carried out for the read circuits as the performance of the read circuits under variations must be explored for the accuracy of the algorithm. For the Monte Carlo simulations carried out we used the Latin hypercube statistical option which divides the parameter subspace into a predefined number of spaces and corners from each subspace are chosen for simulations. In Fig. 2.20, the measured number of pulses is plotted against the total number of pulses counted. The read circuits based on singled-ended adaptive Schmitt trigger did not perform

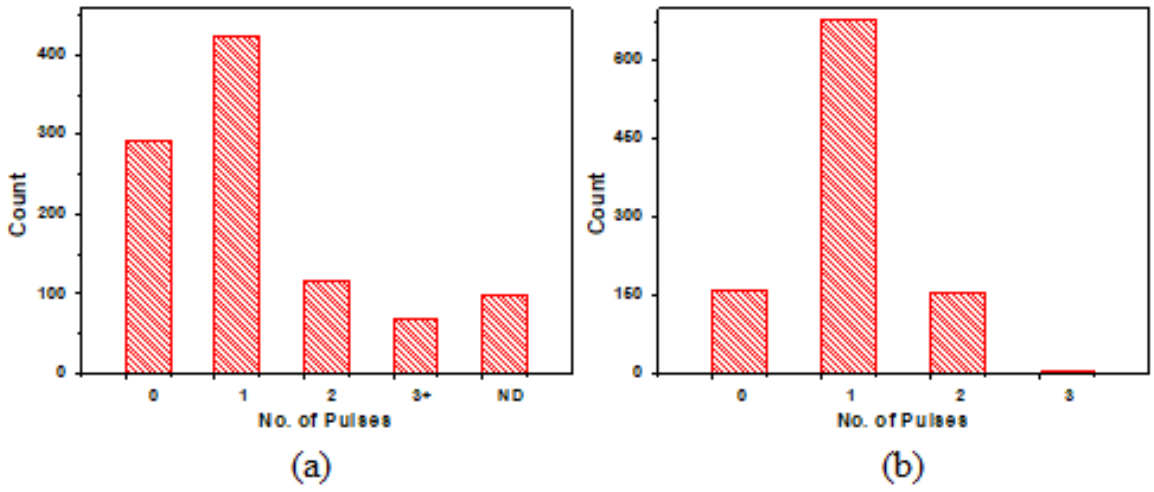



**Figure 2.20:** Results of Monte Carlo simulations which measure the number of pulses against the total count. Both instances correspond to 1 pulse being the correct output. (a) Adaptive read circuit, (b) Differential read circuit.

consistently under process variations since the design is heavily dependent on the size of the transistors. The differential Schmitt trigger based read circuit has a constant voltage reference and hence is less susceptible to variations. We can clearly see that the differential read circuit performs with much improved accuracy against variations. Assuming most of these process variations will be static such as local mismatch, the switching threshold and spike generation circuits could go through a one-time calibration after fabrication.

## 2.5 Demonstration in Learning

We demonstrated the proposed PARCA system on the task of sparse coding, and compared it against a software implementation. MNIST data set LeCun *et al.* (2010) is used to learn the dictionary and extract the image features, with ISTA Daubechies *et al.* (2004) and SGD algorithm Bottou and Bousquet (2008). The software counterpart ran on an Intel Core i7 3.4 GHz 8-core processor. The optimized sparse coding algorithm used in software is obtained from Lin *et al.* (2014), which explores a comparison between Stochastic Coordinate Coding (SCC) and online dictionary learning Mairal *et al.* (2009). The SCC implementation has optimized the computation of

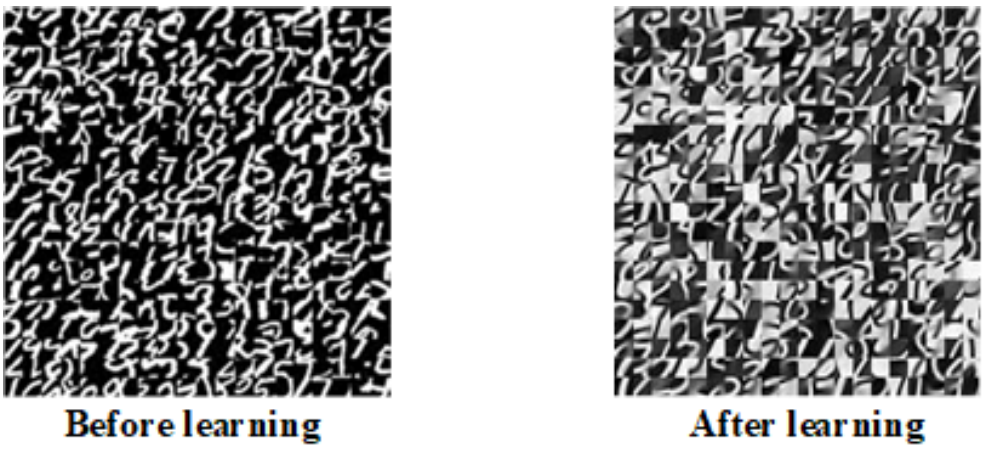



**Figure 2.21:** Demonstration of dictionary learning with MNIST data.

**Table 2.2:** Evaluation of the speedup in computing and energy.

| **Task** | **Software on CPU** | **PARCA** | **Acceleration** |
|---|---|---|---|
| Update $Z$ | 17.2 $m$s (matrix op.) | 4.8 $\mu$s (200 Read) | 3580× |
| Update $D$ | 26.4 $\mu$s (matrix op.) | 336 $n$s (1 Write) | 78× |
| Total Time | 17.2 $m$s | 5.14 $\mu$s | 3340× |
| Total Energy | 208 mJ | 0.8 $\mu$J | – |
| The task above is for one 10×10 image patch, with 100 times in ISTA to update $Z$ and once in SGD to update $D$. | | | |

matrix-vector multiplication and fully uses the sparsity property to further reduce the complexity.

The initial dictionary and the learned dictionary are shown in Fig. 2.21. A good initial condition would enhance the performance of sparse coding since it is a non-convex problem, which indicates that the optimum might not be unique. The initial dictionary used in the paper is simply a random selection among all image patches cropped from the entire data set. Hence the algorithmic performance is not dependent on the quality of its initial condition Coates and Ng (2011).

It can be seen that the learned dictionary well captures local features. Table 2.2 summarizes the computation time and energy consumed by the software and our PARCA system. Both steps of Update $Z$ and Update $D$ benefit from the parallel computing of $D \cdot Z$ ($D^T \cdot r$); Update $D$ is further accelerated by the parallel write of $r \cdot Z$. Overall, PARCA achieves more than 3000× speedup over the software implementation, with higher energy efficiency.

## 2.6 Conclusion

In this paper, we proposed a parallel architecture with resistive crosspoint array for dictionary learning applications, where each dictionary value is represented by the conductance of a RRAM cell. The proposed bio-inspired read circuit converts the RRAM current into digital values in an integrate-and-fire fashion. Using time-dependent synaptic plasticity, the write circuits employ local signals of $Z$ (duty cycle) and $r$ (number of pulses or spikes) to update the conductance of the entire RRAM array in parallel. Peripheral circuits were designed in 65nm CMOS, and simulated with empirical RRAM device models to accelerate computation-intensive tasks in dictionary learning. PARCA demonstrates 3000× acceleration for an image feature extraction task when compared to ISTA and SGD sparse coding software.

For future versions of the PARCA architecture, we intend to explore various alternate algorithms which follow the biological aspect of unsupervised learning. This will pertain the inclusion of inhibition as a form of stabilizing the system, when the system is done learning as the inhibition converges with the positive learning rate. Furthermore, the effect of non-linear update in the conductance of RRAM will be analyzed comprehensively.

Chapter 3

# EFFICIENT MEMORY COMPRESSION IN DEEP NEURAL NETWORKS USING COARSE-GRAIN SPARSIFICATION FOR SPEECH APPLICATIONS

Recent breakthroughs in deep neural networks have led to the proliferation of its use in image and speech applications. Conventional deep neural networks (DNNs) are fully-connected multi-layer networks with hundreds or thousands of neurons in each layer. Such a network requires a very large weight memory to store the connectivity between neurons. In this paper, we propose a hardware-centric methodology to design low power neural networks with significantly smaller memory footprint and computation resource requirements. We achieve this by judiciously dropping connections in large blocks of weights. The corresponding technique, termed coarse-grain sparsification (CGS), introduces hardware-aware sparsity during the DNN training, which leads to efficient weight memory compression and significant computation reduction during classification without losing accuracy. We apply the proposed approach to DNN design for keyword detection and speech recognition. When the two DNNs are trained with 75% of the weights dropped and classified with 5-6 bit weight precision, the weight memory requirement is reduced by 95% compared to their fully-connected counterparts with double precision, while maintaining similar performance in keyword detection accuracy, word error rate, and sentence error rate. To validate this technique in real hardware, a time-multiplexed architecture using a shared multiply and accumulate (MAC) engine was implemented in 65nm and 40nm low power (LP) CMOS. In 40nm at 0.6V, the keyword detection network consumes 36W and the speech recognition network consumes 552W, making this technique highly suitable for mobile and wearable devices.

## 3.1 Introduction

Automatic speech recognition (ASR), which converts words spoken by a person into readable text, has been a popular area of research and development for the past decades Yu and Deng (2016). The goal of ASR is to allow a machine to understand continuous speech in real-time with high accuracy, independent of speaker characteristics, noise, or temporal variations. Nowadays, ASR technology can be easily found in a number of commercial products, including "Google Now" (Google), "Siri" (Apple), and "Echo" (Amazon). In these systems, speech recognition is activated by specific keywords such as "Ok Google", "Hey Siri", and "Alexa". More systems now perform such wake-up keyword detection tasks in an 'always-on' mode, always listening to surrounding acoustics without a dedicated start control. Minimizing the power consumption of such always-on operations that can detect multiple keywords is crucial for mobile and wearable devices. The speech recognition task that follows keyword detection is much more compute-/memory-intensive such that it is typically offloaded to the cloud. A number of commercial systems do not allow speech recognition if the device is not connected to the internet. To expand the usage scenarios, it is important that the speech recognition engine also has low hardware complexity and consumes low power.

One of the widely used approaches for speech recognition employs a Hidden Markov Model (HMM) for modeling the sequence of words/phonemes and uses a Gaussian Mixture Model (GMM) for acoustic modeling Su *et al.* (2010). The most likely sequence can be determined from the HMMs by employing the Viterbi algorithm. For keyword detection, a separate GMM-HMM could be trained for each keyword Rohlicek *et al.* (1989); Wilpon *et al.* (1991), while the out-of-vocabulary (OOV) words are modeled using a garbage or filler model. In recent years, employing

DNNs in conjunction with HMM models for keyword detection and speech recognition have shown substantial improvements in classification accuracy Deng *et al.* (2013b,a). Other prior works in ASR feature recurrent neural networks Graves *et al.* (2013) or convolutional neural networks Sainath and Parada (2015).

Although enhanced accuracy in word detection and recognition algorithms have been demonstrated, implementing such DNNs in hardware requires a large memory footprint and high computation power, due to the large number of parameters (100s of thousands for keyword detection, millions for speech recognition). Considering that mobile and wearable devices have constraints on embedded memory and computational resources and majority of the memory of DNNs comes from the weights between neurons, reducing the number of weight parameters without affecting the accuracy is a necessity for efficient hardware implementation. A number of prior works exist in weight memory reduction for DNNs, including weight and node pruning He *et al.* (2014), fixed-point representation Shah *et al.* (2015), selective weight approximation Venkataramani *et al.* (2014), and quantized weight sharing with Huffman coding Han *et al.* (2015a). Most of these approaches reduce or drop weights on an element-by-element basis, and thus the resulting memory generally cannot be efficiently compressed onto hardware. This is because the address information of which weights are dropped need to be stored as well, which can offset the savings achieved from the reduced number of weights.

In this paper, we propose a coarse-grain sparsification technique on the weights of DNNs, which enables efficient compression of weight memory in an ASIC implementation. The key idea is that we drop the weight connections in large blocks (i.e., $64\times64$), and we enforce the coarse-grain weight drop in a static manner throughout the entire process of training. As a result, the final set of weights can be efficiently mapped onto SRAM arrays with minimal address information for classification. Upon training ini-

tialization, large blocks are randomly dropped with a certain probability (i.e., 75%), and only the remaining weights are subject to training. This is in contrast to Dropout Venkataramani *et al.* (2014) or DropConnect Wan *et al.* (2013) techniques that are commonly used in deep learning algorithms to prevent overfitting. These techniques drop nodes or weights in a dynamic manner using a different random selection in every training iteration. Since the weights that are dropped are different each time, the weight size cannot actually be reduced, and the fully-connected matrix weights are required for the classification phase. On the other hand, partially connected neural networks that have sparse connection matrices show better storage per connection than its fully-connected counterpart and the storage per connection increases as the weight connections become sparser and more random Canning and Gardner (1988).

We introduce an algorithm-hardware co-design scheme for coarse-grain sparsification in order to significantly reduce the memory footprint as well the number of computations. The proposed coarse-grain sparsification technique was evaluated by training DNNs for keyword detection and speech recognition using the DARPA Resource Management (RM) database Price *et al.* (1988) and also by hardware implementations of the corresponding DNNs in 65nm LP and 40nm LP CMOS technology. Overall, this paper makes the following contributions.

1. We propose static coarse-grain sparsification that can substantially reduce the memory footprint as well as necessary computations when the DNNs are implemented onto hardware, with minimal degradation in accuracy.

2. Using the proposed coarse sparsification scheme throughout training and classification, weight memory is compressed by 4× for keyword and speech recognition DNNs. Employing low-precision of 5-6 bits for weights leads to an additional reduction of ∼5×. Overall, the weight memory of the sparse low-precision DNNs

is compressed by ∼20× from that in fully-connected floating-point DNNs.

3. The proposed scheme is implemented and evaluated in 65nm and 40nm CMOS. The DNN implementation of the keyword detection network resulted in 44.80W and 35.38W power consumption in 65nm LP and 40nm LP CMOS, respectively. The speech recognition network is larger and had higher power consumption of 1.07mW and 551.06W in 65nm LP and 40nm LP CMOS, respectively.

The remainder of the paper is organized as follows. In Section 3.2, we describe the DNN processing operations for the two speech applications. Section 3.3 describes the proposed coarse-grain sparsification technique and how the training and classification is performed. In Section 3.4, the architecture of the custom deep neural network hardware with memory compression is described. Section 3.5 presents the experimental results in 65nm LP and 40nm LP CMOS, and the chapter is concluded in Section 3.6.

## 3.2 DNN Processing in Speech Applications

### *3.2.1 DNNs for Keyword Detection and Speech Recognition*

We designed two separate DNNs for keyword detection and speech recognition tasks, which are illustrated in Fig. 3.1. The keyword detection DNN is similar to that in Shah *et al.* (2015) and consists of two hidden layers with 512 neurons per layer (Fig. 3.1(a)). There are 403 input nodes corresponding to 31 frames (15 previous, 15 future and 1 current) with 13 Mel frequency coefficients (MFCC) features per frame. The output layer consists of 12 nodes, where 10 nodes are for the 10 selected keywords, 1 node is for Out of Vocabulary (OOV) words and 1 node is for silence. The output of the neural network represents the probability estimate of each of the nodes in the output layer. These probabilities are calculated using the softmax activation

function.

The DNN designed for speech recognition system, as shown in Fig. 3.1(b), consists of 4 hidden layers with 1,024 neurons per layer. There are 440 input nodes corresponding to 11 frames (5 previous, 5 future and 1 current) with 40 fMLLR (feature-space Maximum Likelihood Linear Regression) features per frame. The output layer consists of 1,483 probability estimates that are sent to the Hidden Markov Model (HMM) unit to determine the best sequence of phonemes. The transcription of the words and sentences for the particular set of phonemes is done using the Kaldi toolkit Povey *et al.* (2011).

### *3.2.2 Training DNNs*

Training the neural network is performed by minimizing the cross-entropy error, as described in Eq. (3.1).

$$E = -\sum_{i=1}^{N} t_i * \ln(y_i) \tag{3.1}$$

Here $N$ is the size of the output layer, $y_i$ is the $i^{th}$ output node and $t_i$ is the $i^{th}$

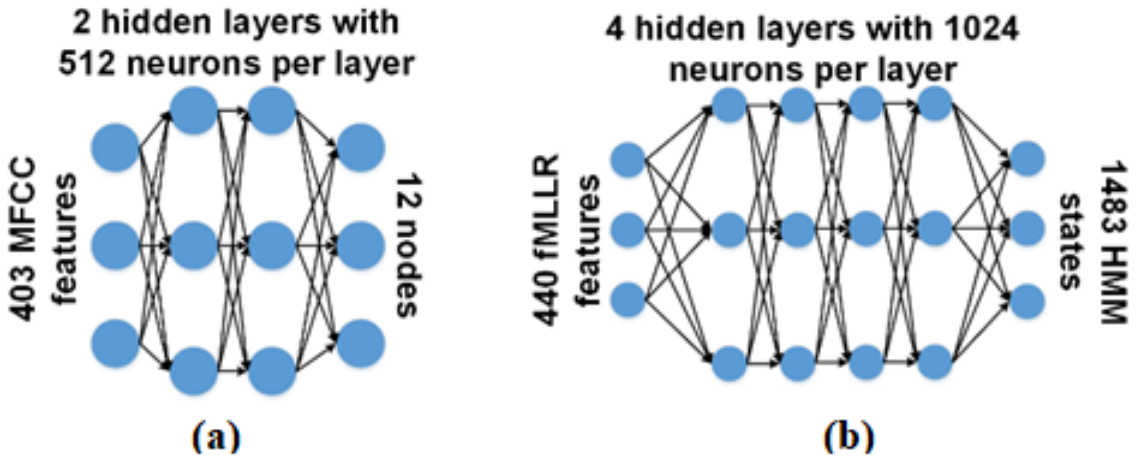



**Figure 3.1:** DNNs for (a) keyword detection and (b) speech recognition.

target value or label. The mini-batch stochastic gradient method Gardner (1984) is used to train the network. The weight $W_{ij}$ is updated in the $(k+1)^{th}$ iteration, using Eq. (3.2).

$$(W_{ij})_{k+1} = (W_{ij})_k + C_{ij} * lr * \{(\Delta W_{ij})_k + \ m * (\Delta W_{ij})_{k-1}\} \tag{3.2}$$

$C_{ij}$ is the binary connection coefficient between two subsequent neural network layers, which is introduced for the proposed CGS, and only the weights in the network corresponding to $C_{ij}$=1 are updated. $m$ is the momentum and $lr$ is the learning rate. The change in weight for each iteration is the differential of the cost function with respect to the weight value, as shown in Eq. (3.3).

$$\Delta W = \frac{\delta E}{\delta W} \tag{3.3}$$

### *3.2.3 Feedforward Classification of DNNs*

Once training is completed, feed-forward computations are performed in the two DNNs as follows. Let the input layer be denoted as $x_i$ ($i = 1,\ 2,\ \ldots\ , N$), where $N$ is the number of input features. The computation in the first hidden layer $h^1$ is given in Eq. (3.4).

$$z_j^1 = \sum_{i=1}^{N} C_{ij} W_{ij}^1 x_i + b_j^1 \tag{3.4}$$

where $z$ is the output neuron, $W$ is the weight matrix and $b$ is the bias for the hidden layer. For both networks, the non-linear activation function that is applied at the end of each hidden layer is ReLU (Rectified Linear Unit) Krizhevsky *et al.* (2012). Other hidden layer neurons are computed similarly.

## 3.3 DNN Weight Compression Using Sparsity and Precision Reduction

In this section, two methods that contribute to the overall weight memory reduction are discussed. First, coarse-grain sparsification (CGS) of fully-connected weight matrix is described for both training and classification of DNNs. In addition, to further reduce the memory footprint during classification, the precision of weights is minimized while maintaining the word detection and recognition accuracy.

### *3.3.1 Coarse-Grain Sparsification (CGS)*

A considerable volume of literature exists on the so-called partially connected neural networks Canning and Gardner (1988), where weight connections between layers are dropped. Previous approaches drop weights based on some specific criteria (e.g., weight value is lower than a threshold). This is performed primarily on an element-by-element basis, which prevents efficient mapping onto hardware memory arrays. To overcome this inefficiency, we propose to randomly drop weights in DNNs in a coarse-grained block-by-block basis.

The fully-connected weight connections in DNNs are first divided into a number

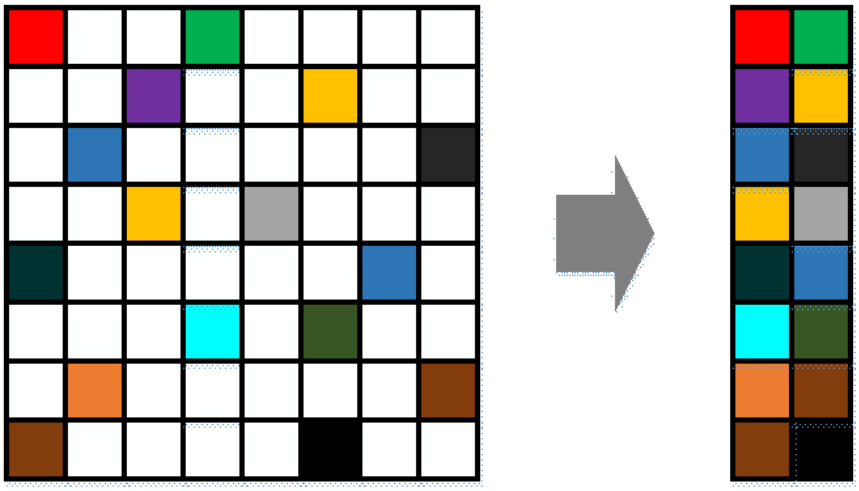


**Figure 3.2:** DNNs for (a) keyword detection and (b) speech recognition.

of large square blocks (e.g., 64×64), and a number of blocks are randomly dropped with a certain probability (e.g., 50%, 75%) prior to training. Throughout training and classification, the dropped blocks remain at zero and do not contribute to the physical memory footprint. In this paper, we focus on power/area reduction for classification hardware, but we postulate that CGS will also lead to low-power acceleration for training.

A sample weight matrix of size 512×512 is shown in Fig. 3.2(a), where the weight matrix is broken down in blocks of size 64×64. In this example, 8 blocks exist along any given row or column. The colored squares represent regions where eligible connections are present, and the white squares represent regions with absence of connections. Note that dropping all blocks along a column or row is not allowed, to prevent any neuron between two layers from being completely disconnected. Fig. 3.2(b) illustrates the compressed (along the row) coarse-grained sparse matrix that only stores regions with active connections.

If the DNN training is done using fully-connected weights, it is much more difficult to sparsify or compress the memory during classification. Therefore, we investigate constraining the weight matrix during training, such that only a small portion (25% in this specific application) of blocks is subject to update. This way, once the training is completed, it becomes straightforward to store only a small number of blocks (25%) with non-zero weights in a compressed form.

Finding the optimal block size that balances hardware overhead and detection / recognition accuracy is crucial. For the same weight drop rate, if the block size is too small, the number of non-zero blocks becomes large, which requires substantial amount of address information and additional combinational logic. On the other hand, if the block size is too large, the training performance suffers due to the inability to represent fine-grain weight connectivity. Orthogonal to the block size, the drop

rate is an important design parameter that directly dictates the maximum allowable compression of memory. We investigated the effect of drop rate and block size independently for keyword detection and speech recognition DNNs. Through simulations, we found that 75% weight drop rate still yielded sufficient accuracy for both keyword detection and speech recognition tasks; the results are presented in Section 3.5.

### *3.3.2 Low-Precision Classification*

The aforementioned training procedure in Section 3.2.2 is performed on a GPU using weights with floating-point representation. Although more recent works investigated low-precision training Courbariaux *et al.* (2015); Gupta *et al.* (2015), combining such low precision in the training phase remains as future work. Once our DNN training is completed off-line, for classification, we represent the weights as well as neurons using a low-precision fixed-point format to further reduce the memory footprint as well as computation cost. For the trained DNN, we first quantize the weights with floating-point neurons down to the precision where the DNN accuracy is acceptable. Then, we operate the DNN with reduced precision on the weights and choose reduced precision for the neurons, while still achieving acceptable accuracy. This methodology most likely leads to a higher precision for the neurons compared to that of the weights, but this is an acceptable trade-off since weights have a larger impact on the overall memory footprint as well as power consumption. Throughout the paper, we denote fixed-point precision using a Q$A$.$B$ format, where $A$ denotes the number of bits assigned to the integer part and $B$ denotes the number of bits assigned to the fractional part. Unless mentioned otherwise, an additional sign bit is assumed.

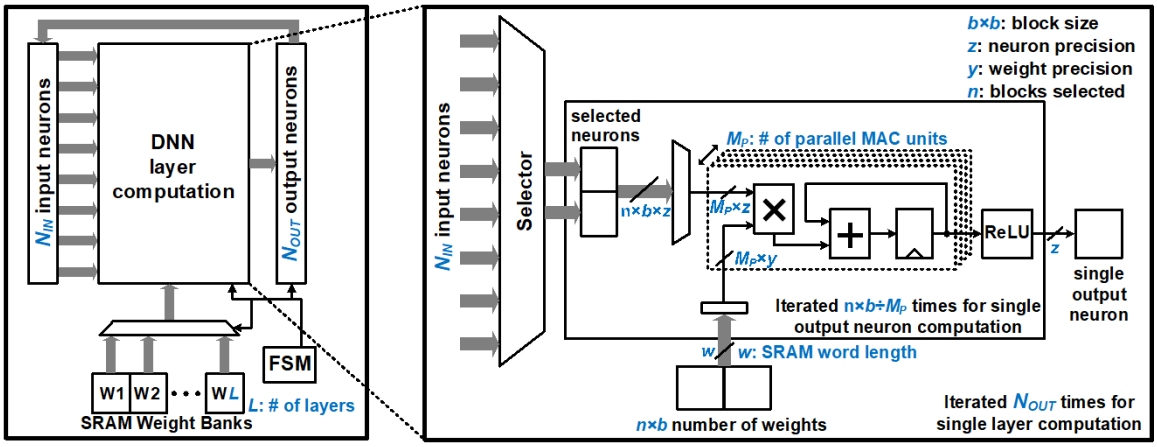



**Figure 3.3:** Block diagram of the DNN-based classification system.

## 3.4 Hardware Architecture and CMOS Implementation

The architecture of the CGS based keyword detection and speech recognition DNNs is shown in Fig. 3.3. The main computation block (shown on the right) operates on one layer at a time. It consists of $M_p$ MAC units that operate in parallel on data that is forwarded by the selector. The weights of the network are stored in $L$ SRAM banks while the neuron inputs and outputs are stored in registers. The finite state machine (FSM) coordinates the data movement and re-use of the computation blocks.

### *3.4.1 Keyword Detection DNN Implementation*

The keyword detection network described in Section 3.2.1 has 403 input neurons and 512 neurons in the two hidden layers, which all have 16-b precision. The network was trained such that the CGS block size is the same for all the layers. The block size used for keyword detection is 64×64, hence the 512 input neurons of the compute block are divided into 8 blocks with 64 inputs per block. Due to the sparsity of the weight matrix, there exists only 128 non-zero weight values with 5-bit precision. The Selector module chooses two blocks out of the eight input blocks and directs them to

the MAC unit. The Selector is implemented using two 8:1 block multiplexers, each of which is controlled by a 3-bit select signal. The select signal contains the spatial location of the blocks in the uncompressed weight matrix and is used to route the data to the MAC. For the keyword detection DNN, the parameters in Fig. 3.3 are $N_{IN} = 512$, $b = 64$, $z = 16$, $y = 5$ and $n = 2$.

The MAC unit operates on inputs from the SRAM that contains stored weights, and from the Selector module that forwards the neuron values corresponding to the chosen weights. After investigating architectures with different number of MAC units, we chose an implementation with 16 parallel MAC units ($M_p = 16$) as it consumes the least amount of power, as shown in Section **5.2**. The MAC unit for the keyword system computes eight sum of products in parallel. After the sum of products computation is completed, ReLU activation is performed, and then conveyed to the output. The output of the MAC unit is a single output neuron, which is used as an input to the next layer's computations. The 4 parallel MAC architecture takes 4,096 cycles to compute on one layer of the keyword detection network.

### *3.4.2 Speech Recognition DNN Implementation*

We employed the same architecture shown in Fig. 3.3 for the speech recognition system. The speech recognition DNN described in Section 3.2.1 has 440 input neurons and 1,024 neurons in the next four hidden layers, all of which have 16-b precision. The network was trained to allow re-use of the computation blocks for all the layers. The block size used for speech recognition is 64×64, hence the 1,024 input neurons of the compute block is divided into 16 blocks of 64 inputs. The Selector module chooses 4 out of the 16 input blocks and directs them to the MAC units. For the speech recognition DNN, the parameters in Fig. 3.3 are $N_{IN} = 1{,}024$, $b = 64$, $z = 16$, $y = 6$ and $n = 4$.

The MAC unit for the speech recognition system receives 256 inputs from the SRAM and Selector blocks. After investigating architectures with different number of MAC units, we chose an architecture with 20 MAC units ($M_p = 20$) since it has the lowest power consumption. The number of cycles required to finish the computation for a layer with 20 MAC units is 13,108 cycles.

### *3.4.3 Finite State Machine*

The finite state machine (FSM) has 5 and 7 distinct states for the keyword and speech system, respectively. In state 0, all the functional blocks are inactive as the weights are loaded onto the SRAM and the system is waiting to compute. Other states control different steps of the computation of the deep neural network layers. For instance, it counts the number of total computations performed by the functional blocks and accordingly feeds it the correct data.

## 3.5 Experimental Results

In this section, we present the software and hardware evaluation results for keyword detection and speech recognition. We compare the performance of the proposed sparsely connected network architecture with fully-connected fixed-point and double-precision floating-point architectures. All software evaluations were conducted on the RM database Price *et al.* (1988). The hardware logic and memory are implemented using TSMC 65nm LP and 40nm LP CMOS technology. The supply voltage of both the memory and logic of DNNs is scaled down to 0.7V (65nm) and 0.6V (40nm) to achieve enhanced power efficiency. The software and hardware evaluation setups for the two networks are described in Section 3.5.1.

### *3.5.1 Experimental Setup*

#### Software Evaluation Setup

For the keyword detection system, there are cases when the system predicts a keyword that is present (true positive) or the system predicts a keyword that is absent (false alarm). A good metric to determine the performance of the system is given by area under the receiver operator characteristics (ROC), referred to as the area under the curve (AUC) Bradley (1997). We evaluate different neural network architectures with respect to their AUCs. The training of network is done using a learning rate of 0.001, momentum of 0.8 and batch size of 500. The network is trained for 6 epochs.

For the speech recognition system, the 40 fMMLR features are extracted from the speech waveform by applying transforms to the MFCC features as described in Rath *et al.* (2013). The fMLLR features of the 5 previous frames and 5 future frames are added to obtain a 440-dimension feature vector for each frame. The baseline GMM-HMM model is trained using the script provided for the RM database. The DNN training is performed using the PDNN toolkit Miao (2014) on the GTX 650 GPU. For training this network, we use the "newbob" technique using an initial learning rate of 0.08, momentum of 0.5 and batch size of 256. If the validation error between two epochs differs by less than 0.2%, the learning rate is scaled by a factor of 0.5. This scaling is performed at every subsequent epoch until training is complete. The training is stopped when the validation error between two consecutive epochs differs by less than 0.2%.

#### Hardware Evaluation Setup

The power consumption results of the compute logic are obtained from PrimeTime simulation of the synthesized functional blocks with data switching activity informa-

tion, while the SRAM power computations are calculated with read current values in the commercial memory compiler datasheet for 65nm LP and 40nm LP CMOS. The power consumption values are first obtained at the nominal voltage and temperature of 1.2V/1.1V (65nm/40nm) and 25°C, and then the scaling ratios are applied to calculate the power and performance at supply voltages down to 0.7V/0.6V (65nm/40nm), which are assumed as the minimum operating supply voltage ($Vmin$) of the 6-T SRAM arrays.

To scale the supply voltage, we use different approaches for logic and memory. For logic, we conduct SPICE simulations to get the voltage scaling ratio for both technology nodes. We obtain this ratio by extracting the reduction in output frequency from simulation results of a 7-stage ring oscillator (RO) that consists of FO4 inverters at various supply voltage values. For 65nm LP and 40nm LP, the RO is simulated with the foundry transistor models. With the information on reference RO frequency and power at corresponding technology nodes, we scale the voltage down such that timing slack of the critical paths in keyword detection and speech recognition design decreases correspondingly. For memory active power, we use the same scaling factors from the RO simulation as done in logic. For memory leakage power, we simulate the leakage power of a SRAM bitcell from nominal voltage down to 0.7V/0.6V, and apply the leakage scaling ratio to project the leakage of SRAM arrays for different supply voltages.

### *3.5.2 Keyword Detection DNN*

**Software Results**

Fig. 3.4 shows the effect of block size and percentage of the dropped connections on the AUC performance of the floating-point keyword detection DNN. The percentage

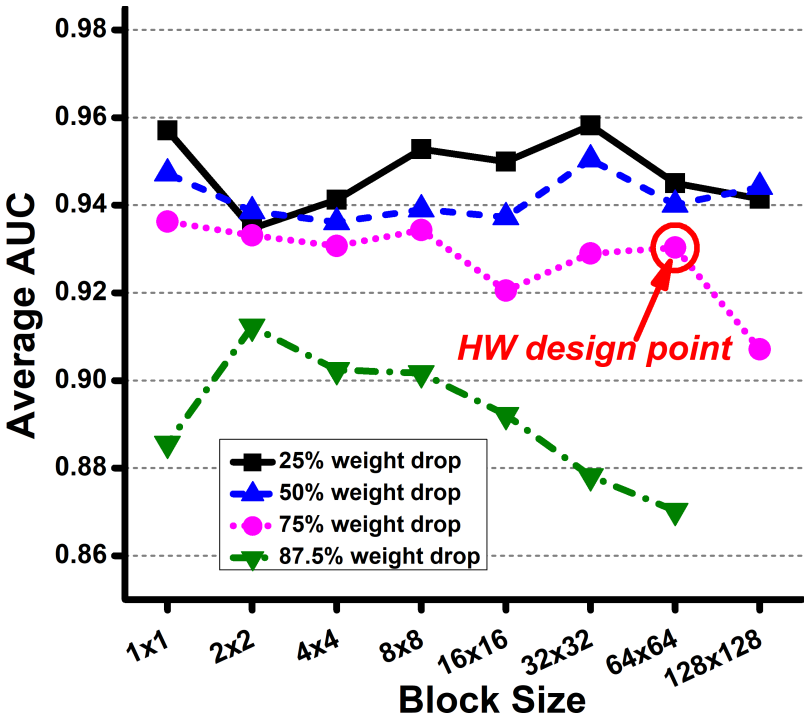



**Figure 3.4:** Effect of block size and percentage drop on average AUC of keyword detection DNN.

of the dropped connections applies for the weights of the two hidden layers. We do not drop connections in the last layer since it consists of only 12 nodes and is relatively sensitive to detection accuracy. In addition, the weights of the last layer contribute to ∼1% of the total weights in the system, thus any reduction in this layer does not result in substantial reduction in the overall system memory requirement. From Fig. 3.4, we observe that, for the same block size, increasing the percentage of dropped connections adversely affects the AUC performance, as expected. When the drop in connection is less than 50%, there is little change in the AUC performance even when the block size is large. However, for larger drop rates, the AUC performance becomes sensitive to the block size. For instance, the performance of a system with 75% of its weights dropped has an AUC performance loss of up to 0.029 when the block size is 128×128. Since the AUC performance loss is only 0.015 when the block size is 64×64, this configuration is selected as the hardware design point for our sparse network.

**Table 3.1:** Comparison of memory requirements and AUC for keyword detection networks.

| **Architecture** | **Memory size** | **Average AUC** |
|---|---|---|
| Floating-point (fully-connected) | 1.81 MB | 0.945 |
| Fixed-point (fully-connected) | 290.32 KB | 0.940 |
| Proposed CGS (block size: 64x64, 75% weight drop) | 84.38 KB | 0.912 |

For the fully-connected fixed-point architecture, the weights and biases are represented using Q2.2 format. The inputs and hidden layers are represented by 15 bits in Q2.13 and Q10.5, respectively, which correspond to 16 bits when the sign bit is included. We use the same set of precisions for the proposed sparse architecture with block size of 64×64 and 75% dropout.

Table 3.1 compares the memory requirements and the performance of the system for different keyword detection networks. We see that there is a small drop in the performance of proposed CGS scheme compared to the fully-connected fixed-point and floating-point architectures. The ROC curves for the three different architectures shown in Fig. 3.5 are largely similar. From these results, we conclude that our coarsely sparsified DNN performs at a level similar to the floating-point architecture, while requiring only 4.13% of the memory required by the fully-connected floating-point DNN.

### Hardware Results

The keyword detection system contains three SRAM memory banks: two 40KB banks for the hidden layer, and one 3.75KB bank for the final layer. The system must operate at a minimum of 1.56 MHz to meet the latency requirement of 10ms per input (for a single MAC architecture).

Fig. 3.6 shows the design space exploration of the keyword detection network,

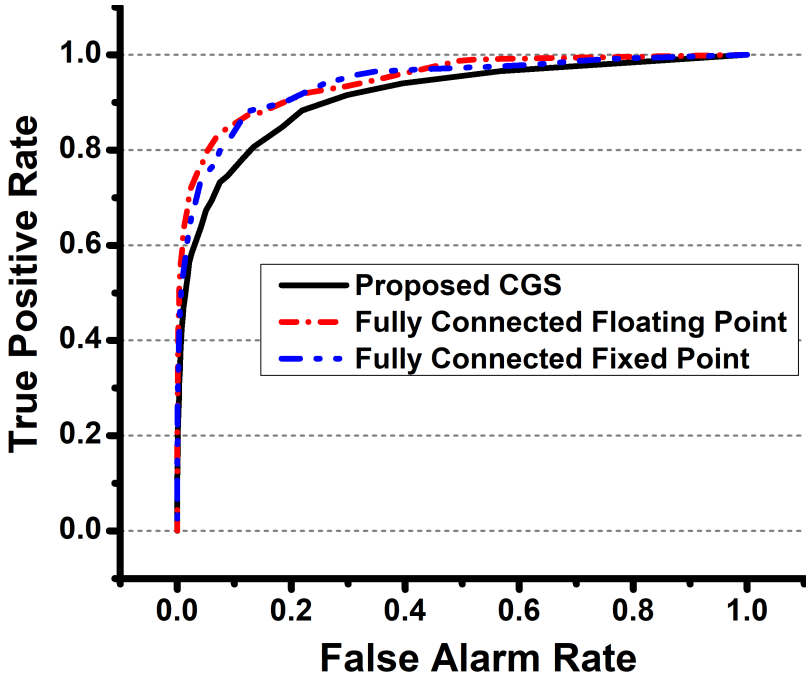



**Figure 3.5:** ROC Curve of different implementations for keyword detection DNN.

which depicts the change in power consumption and logic area as a function of the number of MAC units. The increase in the number of MAC units allows for reducing the operating frequency, which in turn increases the timing slack of critical paths. The increase in slack is compensated by scaling down the supply voltage resulting in reduction in power. Also the 4× reduction in computation facilitated by the proposed CGS enables keyword classification in real-time. For CMOS technology nodes, it is observed that the lowest power consumption is obtained when the number of parallel MAC units is 16. Increasing the number of parallel MAC units beyond that requires a SRAM size larger than the total weights of the keyword detection DNN weights, leading to increase in total power consumption.

During the neural network computations, as only one weight bank performs read operation, the other two are kept in standby mode consuming leakage power. Overall, the power consumption is 44.80W when operating at 0.7V for 65nm LP and 35.38W at 0.6V for 40nm LP CMOS. Table 3.2 lists the area and power contributions of

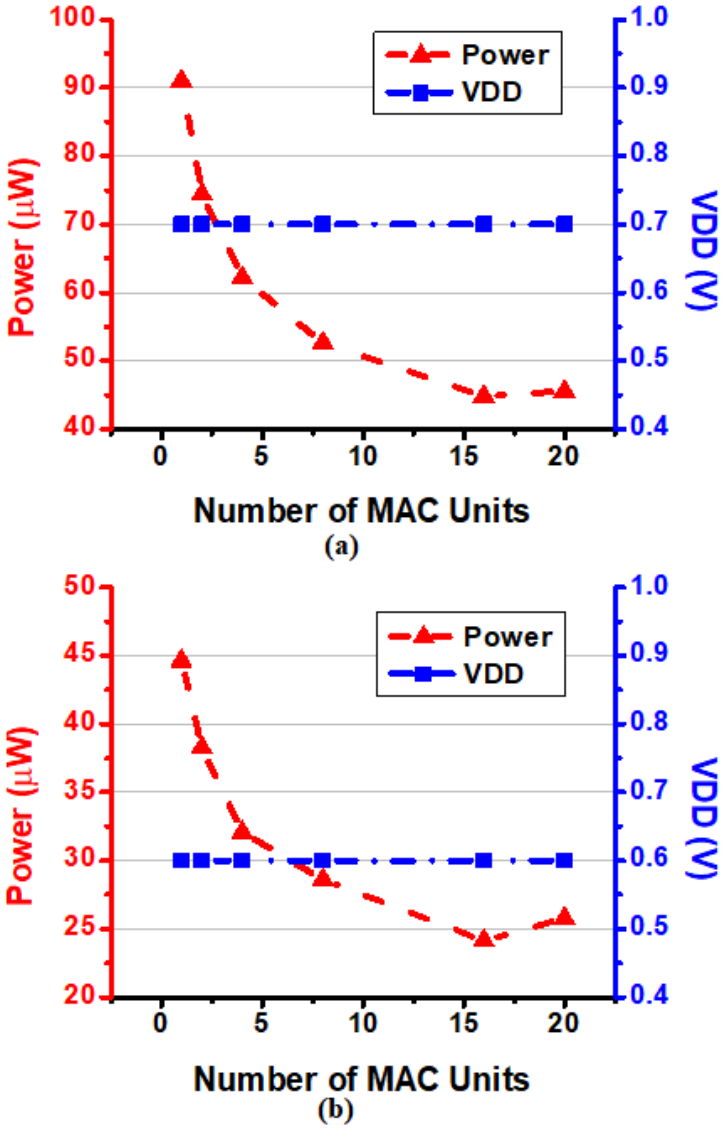



**Figure 3.6:** For the same real-time performance, supply voltage, system power, and area results for keyword detection DNN are shown for different number of parallel MAC units in (a) 65nm LP and (b) 40nm LP CMOS.

the 16 MAC architecture. We also observe that the SRAM power consumption dominates, consuming >70% of the total system power, and that the SRAM leakage power consumes a higher percentage of the total power in 40nm, compared to that in 65nm.

**Table 3.2:** Power and area for keyword detection networks.

| 2* | **Area ($\mu$m$^2$)** | | **Power ($\mu$W)** | |
|---|---|---|---|---|
| | **65nm** | **40nm** | **65nm** | **40nm** |
| Selector | 38,045 | 13,592 | 7.66 | 2.53 |
| Compute Unit | 12,524 | 5,233 | 5.73 | 2.06 |
| SRAM mux, output demux, FSM | 32,682 | 14,372 | 11.79 | 6.65 |
| SRAM (leakage) | 882,414 | 472,418 | 19.62 (4.40) | 24.14 (5.19) |
| **Total** | **965,665** | **505,615** | **44.80** | **35.38** |

### *3.5.3 Speech Recognition DNN*

**Software Results**

The performance of the speech recognition system is measured by two key metrics, the word error rate (WER) and the sentence error rate (SER). The word error rate is defined as WER=100*(S+D+I)/N, where $S$ denotes the number of substitutions, $D$ is the number of deletions, $I$ is the number of insertions and $N$ is the number of words in the reference. The sentence error rate is the percentage of sentences that has at least one error. While WER and SER are for the whole system (DNN+HMM), here we only analyze the effect of the DNN parameters on the error rates. Fig. 3.7 shows the effect of percentage of the dropped connections on the WER of the floating point system as a function of the block size. For up to 75% percentage of dropped weights at all layers of the network, the WER performance of the system is comparable to fully-connected floating-point DNNs. Increasing the drop rate to 87.5% for block sizes larger than 64×64 worsens the error rate significantly, and is not preferable. Based

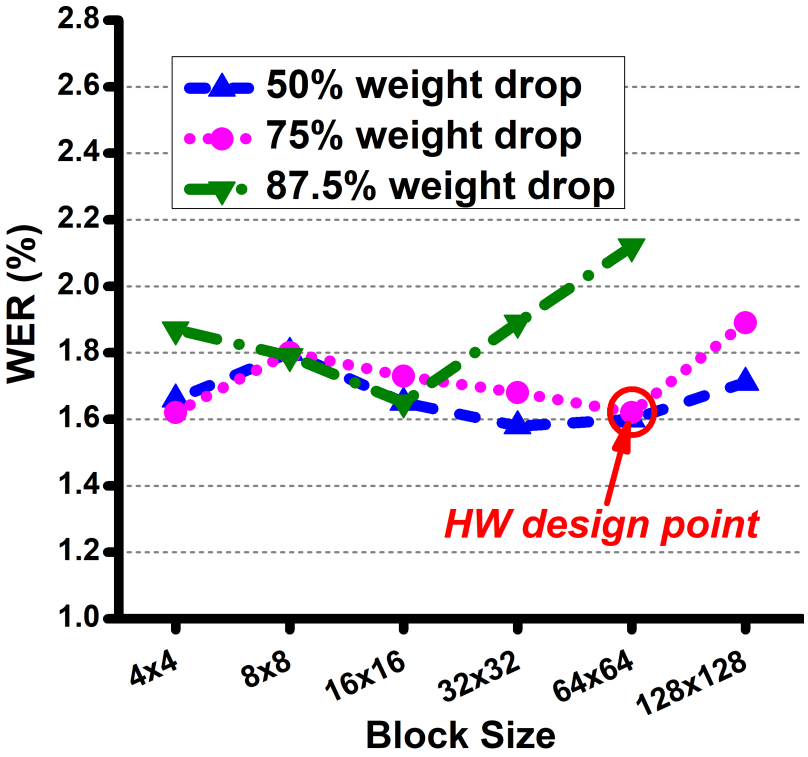



**Figure 3.7:** For the same real-time performance, supply voltage, system power, and area results for keyword detection DNN are shown for different number of parallel MAC units in (a) 65nm LP and (b) 40nm LP CMOS.

on this analysis, we choose a drop rate of 75% across all layers with block size of 64×64.

In order to derive a fixed-point speech recognition DNN, we follow the same procedure as the keyword detection DNN. The histogram of the weights and biases is used to determine the precision of the integer bits and fractional precision so that the error rates are below 2%. In this case, the fractional precision was 5 bits and the weights are represented by Q0.5. The precision of the hidden layers was found to be Q10.5, and the input neurons are represented by Q4.11. Therefore, both keyword detection and speech recognition DNNs require 16 bits for the hidden layer nodes.

Table 3.3 compares the performance of our system with the fully-connected floating-point and fixed-point architectures. The sparse fixed-point DNN using the proposed

**Table 3.3:** Comparison of memory, SER, and WER for speech recognition networks.

| **Architecture** | **WER** | **SER** | **Memory** |
|---|---|---|---|
| Floating-point (fully-connected) | 1.64% | 10.89% | 19.53MB |
| Fixed-point (fully-connected) | 1.77% | 11.10% | 3.66MB |
| Proposed CGS (block size : 64x64, 75% dropout) | 1.67% | 10.96% | 1.02MB |

CGS technique with up to 75% of its connections dropped, has an WER close to that of the floating point fully-connected DNN. The proposed architecture requires memory size of only 1.02MB compared to 19.53MB of a fully-connected floating-point architecture. Thus, the sparsified fixed-point network is able to reduce ∼95% of the memory requirement with minimal degradation in WER/SER performance.

### Hardware Results

The speech recognition system with 20 MAC units operates at 6.7MHz to meet the 10ms latency requirement in order to function at real time. There are five SRAM banks that store the DNN weights, where the size of each bank is 224KB. Similar to the keyword detection network, only one SRAM bank is active at a time while four other banks will be in standby mode.

Fig. 3.8 shows how the number of parallel MAC units affects voltage/frequency scaling for the same real-time constraint, and thus the power of the speech recognition DNN. For both technology nodes, the 20 MAC unit architecture is found to be the optimal design point in terms of power consumption. The area and power values for the 20 MAC architecture are shown in Table 3.4.

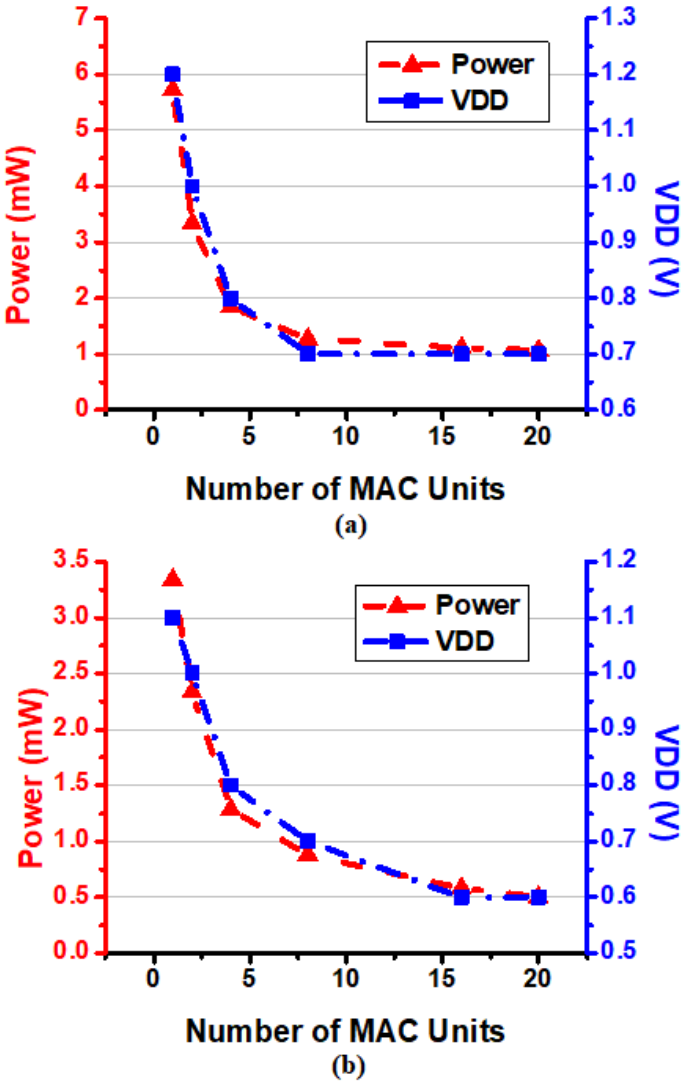



**Figure 3.8:** For the same real-time performance, supply voltage, system power, and area results for keyword detection DNN are shown for different number of parallel MAC units in (a) 65nm LP and (b) 40nm LP CMOS.

Overall, the power consumption of the speech recognition system is only 1.07mW (at 0.7V) in 65nm LP and 551.06W (at 0.6V) in 40nm LP. Similar to the keyword detection DNN, the power of the speech recognition DNN is predominantly due to the SRAM, consuming >80% of total system power. Therefore, at lower voltages, as the frequency scales down with the increase in the number of MAC units, the leakage power consumes a larger portion of the total power, especially in 40nm LP.

**Table 3.4:** Evaluation of the speedup in computing and energy.

| 2* | **Area ($\mu$m$^2$)** | | **Power ($\mu$W)** | |
|---|---|---|---|---|
| | **65nm** | **40nm** | **65nm** | **40nm** |
| Selector | 119,481 | 61,221 | 110.80 | 49.22 |
| Compute Unit | 20,953 | 10,883 | 42.41 | 22.63 |
| SRAM mux, output demux, FSM | 182,264 | 58,624 | 143.54 | 61.24 |
| SRAM (leakage) | 8,333,200 | 4,505,040 | 771.58 (48.90) | 417.97 (59.70) |
| **Total** | **8,655,898** | **4,635,768** | **1,068.33** | **551.06** |

## 3.6 Conclusion

In this paper, we proposed a software-hardware co-design scheme of deep neural networks where the weight memory is compressed by dropping connections in blocks, leading to more compact and low-power hardware. We show that a network with 75% of its connected network performs at a level similar to that of a fully connected network with similar architecture. We test this approach on two speech applications, namely keyword detection and speech recognition on the RM database. We implemented both networks in 65nm and 40nm LP CMOS, exploring the performance and power trade-offs between computation and memory. Using the proposed CGS technique together with the fixed-point precision weights, the total weight memory of DNNs are reduced by ∼95% compared to that of a fully-connected floating-point DNN. Employing voltage scaling, the keyword detection and speech recognition network consumes 35.38W and 551.06W in 40nm, respectively. For future works, we intend to use the CGS architecture on recursive neural networks and convolutional neural networks to achieve comparable power and area savings demonstrated above.

Chapter 4

# POWER, PERFORMANCE, AND AREA BENEFIT OF MONOLITHIC 3D ICS FOR ON-CHIP SPARSE DEEP NEURAL NETWORKS TARGETING SPEECH RECOGNITION

In recent years, deep learning has become widespread for various real-world recognition tasks. In addition to recognition accuracy, energy-efficiency and speed (i.e., performance) is another grand challenge to enable local intelligence in edge devices. In this paper, we investigate the adoption of monolithic 3D IC (M3D) technology for deep learning hardware design, using speech recognition as a test vehicle. M3D has recently proven to be one of the leading contenders to address the power, performance and area (PPA) scaling challenges in advanced technology nodes. Our study encompasses the influence of key parameters in DNN hardware implementations towards their performance and energy-efficiency, including DNN architectural choices, underlying workloads, and tier partitioning choices in M3D designs. Our post-layout M3D designs, together with hardware-efficient sparse algorithms, produce power savings and performance improvement beyond what can be achieved using conventional 2D ICs. Experimental results show that M3D offers 22.3% iso-performance power saving and 6.2% performance improvement, convincingly demonstrating its entitlement as a solution for DNN ASICs. We further present architectural and physical design guidelines for M3D DNNs to maximize the benefits.

## 4.1 Introduction

Deep neural networks (DNNs) have become ubiquitous in many machine learning applications, from speech recognition Deng *et al.* (2013a); Graves *et al.* (2013) and

natural language processing Conneau *et al.* (2017), to image recognition Krizhevsky *et al.* (2012); He *et al.* (2016), and computer vision Karpathy and Fei-Fei (2015). Large neural network models have proven to be very powerful in all the stated cases, but implementing high-speed (i.e., high-performance), energy-efficient DNN ASIC is still challenging because (1) the required computations consume large amounts of processing time and energy, (2) the memory needed to store the weights are prohibitive, and (3) excessive wire overhead exists due to a large number of connections between neurons, which makes a DNN ASIC a heavily wire-dominated circuit.

Modern DNNs may require >100M parameters Xiong *et al.* (2016) for large-scale speech recognition tasks. This is impractical using only on-chip memory due to power density and temperature instability Liao *et al.* (2005) , and hence offloading storage to an external DRAM is required. With the introduction of an external DRAM, however, the bottleneck for computation efficiency is now determined by the parameter fetching from DRAM Sze *et al.* (2016). To mitigate this bottleneck, recent works have compressed the neural network weights *in architectural perspective* and substantially reduced the amount of computation required to obtain the final output He *et al.* (2014); Han *et al.* (2016); Kadetotad *et al.* (2016); Cheng *et al.* (2017), which becomes crucial for efficient DNN ASICs. An alternate method of reducing the complexity caused by the vast requirement of memory for DNNs is in-training quantization of the network parameters Courbariaux *et al.* (2015, 2016), this method however is not explored in the current work.

With the weight-compressed DNN architecture, we adopt monolithic 3D IC (M3D) technology Batude *et al.* (2009) to further improve the energy-efficiency and performance *in physical design perspective*. As device scaling in advanced technology nodes is slowly saturating due to low volume and yields, 3D IC technology has come into the spotlight as an alternative for continuing Moore's low. M3D has shown its strength

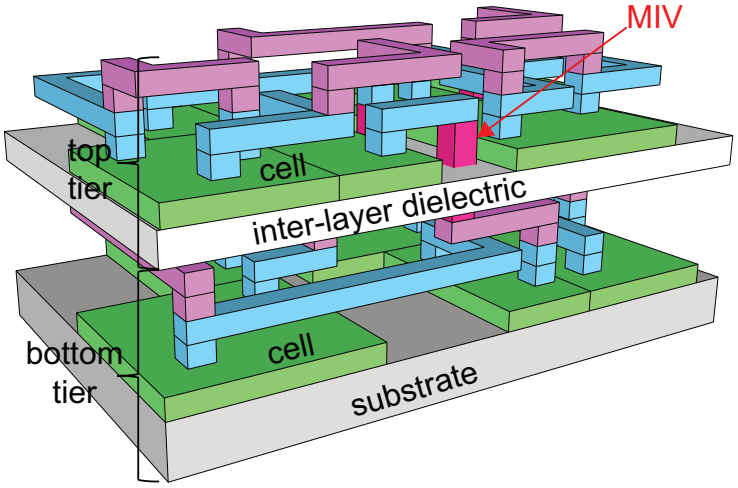



**Figure 4.1:** A schematic showing a gate-level monolithic 3D IC (M3D).

in reducing power consumption and enhancing performance by effectively minimizing wirelength as well as routing congestion, especially in wire-dominated circuits like DNN ASICs. As shown in Fig. 4.1, in M3D, transistors are fabricated onto multiple tiers, and the connections crossing the tiers are established by nano-scale monolithic inter-tier vias (MIVs) Batude *et al.* (2009). Owing to the minuscule size of MIVs ($<100nm$), M3D achieves orders of magnitude denser vertical integration with lower RC parasitics compared with through-silicon via (TSV)-based 3D ICs Nayak *et al.* (2015). In so-called gate-level M3D, each standard cell occupies a single tier—as opposed to being split into multiple tiers—and MIVs are utilized for inter-cell connections that cross tiers. Efficient CAD tool flows exist Panth *et al.* (2014); Chang *et al.* (2016a), and studies have demonstrated its performance and power improvements across multiple technology generations Chang *et al.* (2016b).

In this paper, for the first time, we investigate the benefit of M3D on DNN ASIC implementations and explore architectural and physical design decisions that impact its power consumption Chang *et al.* (2017) and performance. We present two DNN architectures for speech recognition with different granularity of weight compression, and implement them in both 2D and M3D designs. We also examine two schemes for

memory floorplan in M3D designs, and comprehensively compare power, performance and area (PPA) benefits. The main contributions of this paper are as follows: (1) the impact of M3D on DNN architectures with different granularity in sparsity is investigated, (2) we study the impact of tier partitioning in our M3D designs to better handle memory blocks, (3) feed-forward classification and pseudo-training workloads are examined thoroughly to investigate their impact on power reduction. (4) We demonstrate an in-depth analysis on the performance benefit of M3D DNN ASICs over their 2D counterparts, and (5) present key guidelines on optimal architectural and physical design decisions for M3D DNN ASICs.

## 4.2 Deep Neural Network for Speech Recognition

In this paper, we focus on DNN for speech recognition, but the proposed methodologies can be adopted to DNNs for other applications as well. In this section, we present the topology, the training and classification strategy of our DNN architectures, and the coarse-grain sparsification (CGS) which effectively reduces area and computation overhead of DNNs.

### *4.2.1 Our DNN Topology*

Starting from a fully-connected DNN, we adopt a Gaussian Mixture Model (GMM) for acoustic modeling Su *et al.* (2010). Since it has been shown that DNNs in conjunction with Hidden Markov Models (HMMs) increase recognition accuracy Deng *et al.* (2013a), a HMM is also employed to model the sequence of phonemes. The most likely sequence is determined by the HMM utilizing the Viterbi algorithm for decoding. Then, we adopt the CGS methodology presented in Kadetotad *et al.* (2016) in our DNN architecture to reduce the memory footprint as well as the computation for DNN classification.

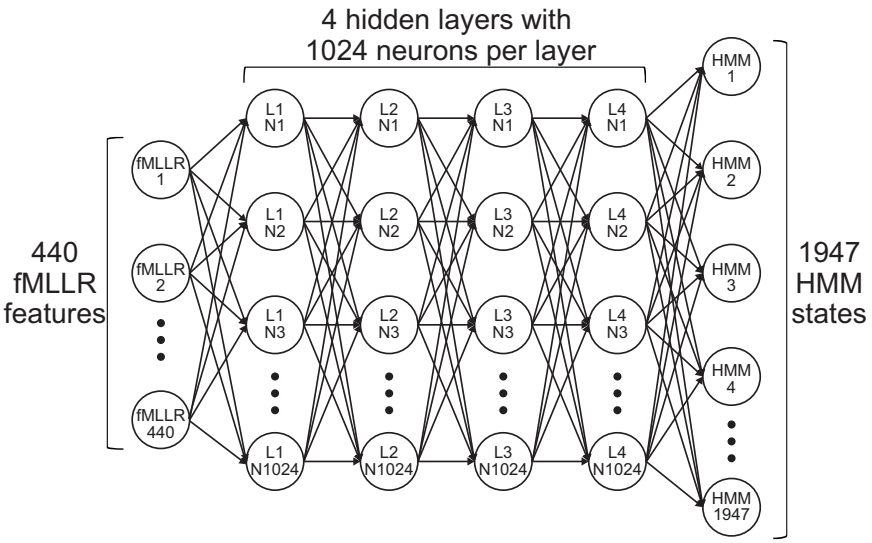



**Figure 4.2:** Diagram of our DNN for speech recognition.

As shown in Fig. 4.2, our DNN for speech recognition consists of 4 hidden layers with 1,024 neurons per layer. There are 440 input nodes corresponding to 11 frames (5 previous, 5 future, and 1 current) with 40 feature-space Maximum Likelihood Linear Regression (fMLLR) features per frame. The output layer consists of 1,947 probability estimates, and they are sent to the HMM unit to determine the best sequence of phoneme using the TIMIT database Garofolo *et al.* (1993). The Kaldi toolkit Povey *et al.* (2011) is utilized for the transcription of the words and sentences for the particular set of phonemes.

### *4.2.2 DNN Training and Classification*

Our DNN is trained with the objective function that minimizes the cross-entropy error of the outputs of the network, as described in Eq. 4.1.

$$E = -\sum_{i=1}^{N} t_i \cdot ln(y_i), \tag{4.1}$$

where $N$ is the size of the output layer, $y_i$ is the $i^{th}$ output node, and $t_i$ is the $i^{th}$ target value or label. The mini-batch stochastic gradient method Gardner (1984) is

used to update the weights. The weight $W_{ij}$ is updated in the $(k+1)^{th}$ iteration using Eq.4.2.

$$(W_{ij})_{k+1} = (W_{ij})_k + C_{ij}(-lr(\Delta W_{ij})_k + m(\Delta W_{ij})_{k-1}), \tag{4.2}$$

where $m$ is the momentum, $lr$ is the learning rate, and $C_{ij}$ is the binary connection coefficient between two subsequent neural network layers for CGS. In CGS, only the weights that correspond to the location where $C_{ij} = 1$ are updated. The change in weight for each iteration is the differential of the cost function with respect to the weight value:

$$\Delta W = \frac{\delta E}{\delta W}, \tag{4.3}$$

such that the loss reduces in each iteration. The training procedure is performed on a GPU with 32-bit floating point values.

After training, feed-forward computation is performed for classification, through matrix-vector multiplication of weight matrices and neuron vectors in each layer to obtain the output of the final layer. The Rectified Linear Unit (ReLU) function Krizhevsky *et al.* (2012) is used for the non-linear activation function at the end of each hidden layer.

### *4.2.3 Coarse-Grain Sparsification (CGS)*

To efficiently map sparse weight matrices to memory arrays, CGS methodology Kadetotad *et al.* (2016) is employed. In CGS, connections between two consecutive layers in a DNN are compressed in a block-wise manner. An example of block-wise weight compression is demonstrated in Fig. 4.3. For a given block size of 16×16, it reduces a 1024×1024 weight matrix to 64×64 weight blocks. With a compression ratio of 87.5%, only eight weight blocks (12.5%) remain non-zero for each block row, thus allowing for efficient compression of the entire weight matrix with minimal index.

CGS, when compared to recent neural network compression algorithms such as in Han *et al.* (2017); Cheng *et al.* (2015), offers simpler hardware implementation through CGS multiplexers and MACs. In Han *et al.* (2017), a complex sparse matrix vector multiplication module is required. On the other hand, the methodology in Cheng *et al.* (2015) offers to reduce the order of computations needed for a matrix of size $n$ to $\mathcal{O}(n \log n)$ and reduce the space required to store the matrix to $\mathcal{O}(n)$. However, there is considerable loss in accuracy when the size of the matrix increases, and hardware for computing Fast Fourier transform (FFT) and Inverse Fast Fourier transform (IFFT) is required. The issue of matrix size is resolved in Liao *et al.* (2017) using block-circulant matrices, but the advantage of using FFT and IFFT to compute matrix vector multiplications is lost if the size of the blocks reduce significantly. This restriction is not present if CGS is used.

GPU-accelerated DNN computations can also benefit from CGS. With CGS, along with the testing inference, training complexity can also be reduced due to the sparse nature of the weight matrices. The structured sparseness allows for writing customized GPU kernels that only need to operate on the non-zero elements, significantly speeding up training and reducing GPU power consumption as shown in Gray *et al.* (2017).

In order to study the impact of M3D on PPA in different DNN architectures, the block sizes are swept for the compression ratio of 87.5%, and the two DNN architectures that have the two lowest phoneme error rates (PER) for the TIMIT dataset are selected for hardware implementation. The two architectures chosen are the DNN with 16×16 block size (DNN CGS-16) and the DNN with 64×64 block size (DNN CGS-64), as shown in Table 4.1.

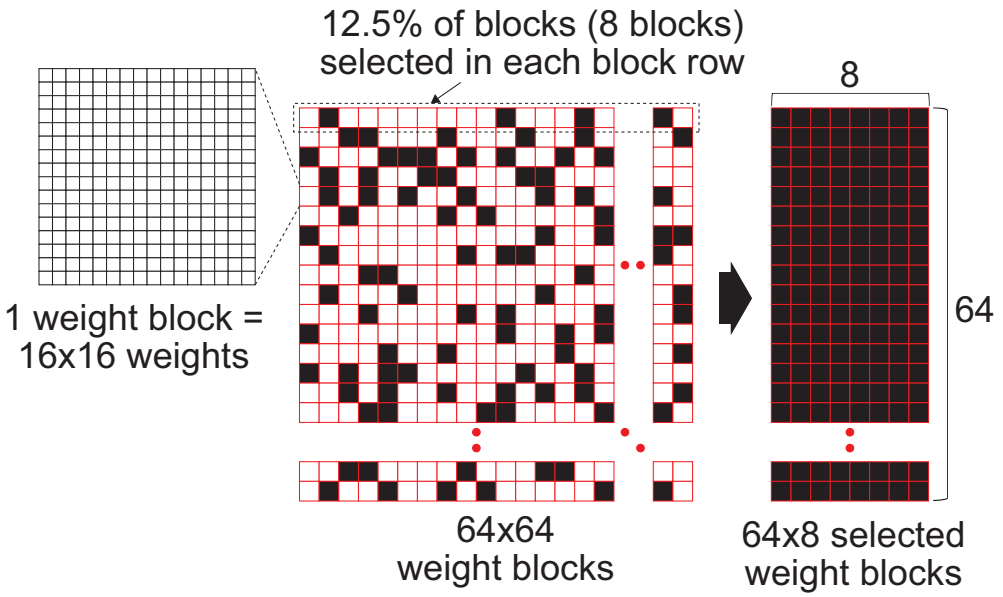



**Figure 4.3:** 1024×1024 weight matrix is divided into 64×64 weight blocks with each weight block having 16×16 weights (i.e. block size of 16×16). 87.5% of weight blocks are dropped using coarse-grain sparsification (CGS). The remaining 12.5% weight blocks are stored in memory.

## 4.3 Full-Chip Monolithic 3D IC (M3D) Design Flow

To implement two-tier full-chip M3D designs of the chosen DNN architectures, we use the state-of-the-art M3D design flow presented in Panth *et al.* (2014). The flow starts with scaling width and height of all standard cells and metal layers by $1/\sqrt{2}$, so that an overlap-free design can be implemented in half the footprint of the corresponding 2D design. The shrunk cells and metal layers are then used to implement a shrunk 2D design by performing all design stages including placement,

**Table 4.1:** Key parameters of the two CGS-based DNN architectures used in our study: block size of 16×16 (DNN CGS-16) and block size of 64×64 (DNN CGS-64).

| parameter | DNN CGS-16 | DNN CGS-64 |
|---|---|---|
| block size | 16×16 | 64×64 |
| compression rate | 87.5% | 87.5% |
| phoneme error rate | 19.8% | 19.9% |

pre-CTS (clock tree synthesis) optimization, CTS, post-CTS optimization, routing, and post-route optimization in Cadence® Innovus™. From this shrunk 2D design, only the cell placement information (x-y location of cells) is retained, and all other information is discarded.

Next, the cells in the shrunk 2D design are scaled back to their original size, resulting in overlap between the cells. In order to remove the overlap, the cells in the shrunk 2D design are partitioned into two tiers. This is accomplished using an area-balanced min-cut partitioning algorithm, which enables half of the cells to be placed on the top tier, and the other half on the bottom tier while minimizing the number of connections between them. The connections between the top and bottom tiers utilize MIVs in the final M3D design. After partitioning, the remaining overlapped cells on both tiers are removed through legalizing.

In order to determine the location of MIVs, we first duplicate all metal layers used in the design, so that the original metal layers represent the metal layers on the bottom tier, and the duplicated layers represent those on the top tier. Then, we define two flavors for all standard cells and memory blocks: the bottom tier cells and the top tier cells. Pins on the bottom tier cells are assigned to the original metal layers, and those on the top tier cells to the duplicated metal layers. After mapping all cells and memory blocks onto their corresponding flavor, the structure is routed in Cadence® Innovus™. The locations of vias between the top metal layer of the original stack and the bottom metal layer of the duplicated stack become MIVs in the final M3D design.

Once the cell and MIV locations are determined, two designs, the top and bottom tier designs, are generated, and trial routing is performed for each tier. Using Synopsys PrimeTime® and trial-routed designs for each tier, timing constraints for both tiers are derived. The timing constrains are used to perform timing-driven detailed

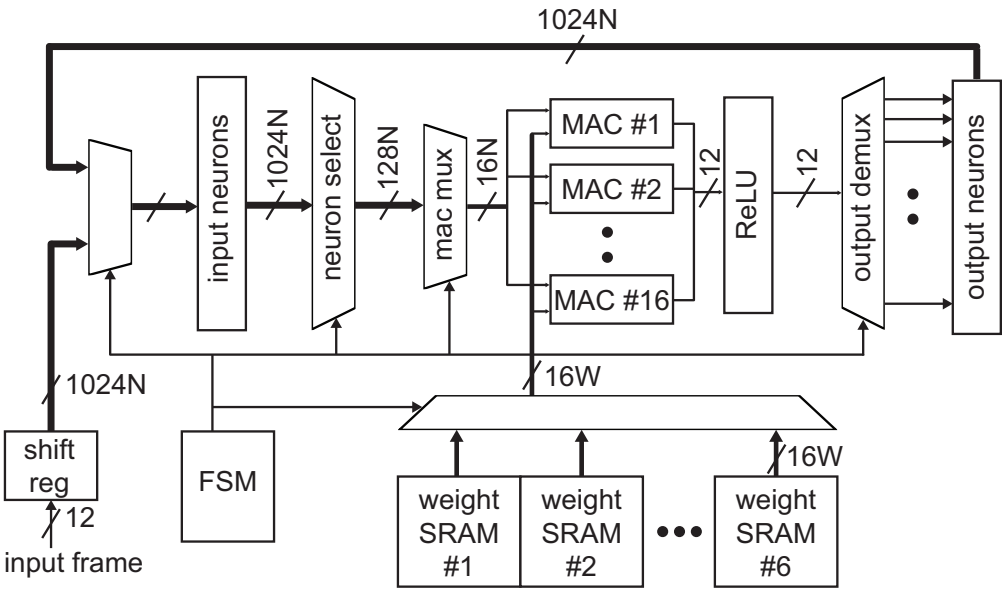



**Figure 4.4:** Block diagram of the proposed CGS-based DNN architecture for speech recognition.

routing for each tier, which results in the final M3D design.

## 4.4 DNN Architecture Description

The block diagram of our CGS-based DNN architecture is shown in Fig. 4.4. The DNN operates on one layer at a time and consists of 16 multiply and accumulate (MAC) units that operate in parallel. The weights of the network are stored in the SRAM banks, while the input and output neurons are stored in registers. The finite state machine (FSM) coordinates the data flow such as layer control and computational resource allocation (i.e. MAC units).

Since the target compression ratio of our architectures is 87.5%, the neuron select unit chooses 128 neurons (12.5%) among 1,024 input neurons that proceed to the MAC units. This selection-based computation eliminates unnecessary MAC operations (i.e., MAC operation of neurons corresponding to zero weights in CGS-based weight matrix). The neuron select unit is controlled by the binary connection coefficients discussed in Section 4.2.2, and the coefficients are stored in the dedicated

register file in the FSM unit.

The size of the register file is determined by the block size used in the DNN architecture. For example, for each hidden layer, eight weight blocks per each row of 64×64 weight blocks are selected for MAC operation in the DNN CGS-16 architecture (Fig. 4.3). Thus, eight multiplexers are required in the neuron select unit, and each multiplexer selects one weight block among 64 in a block row, so that each multiplexer requires six selection bits (=$\log_2 64$). Since there are 64 total block rows in the architecture, the total number of bits to obtain 64×8 selected weight block for a hidden layer is 3,072 bits (= eight multiplexers × 6 selection bits × 64 block rows). Although the DNN has four hidden layers, the number of coefficients for the last hidden layer should be doubled because the number of neurons in the output layer (1,947 HMM states) is almost 2× of other layers. Therefore, the size of the coefficient register file in the DNN CGS-16 is 15,360 bits (= 3,072 bits × 5 effective layers). This value is calculated in the same way for the DNN CGS-64 architecture, resulting in 640 bits in total.

On-chip SRAM arrays store the compressed weight parameters in six banks for the four hidden layers and the output layer (∼2× parameters). The size of the SRAM bank is determined by the number of MAC units in the architecture. Since our DNN architectures operate 16 units in parallel, the row size of each SRAM bank is 128 bits (= 16 MAC units × 8-bit weight precision). Since we assume 8,192 rows for each SRAM bank, the total size of the six SRAM banks in the DNN is 6Mb (= 6 banks × 128 bits × 8,192 rows). This compact memory size with the CGS methodology enables the DNN to store the compressed weight parameters on chip.

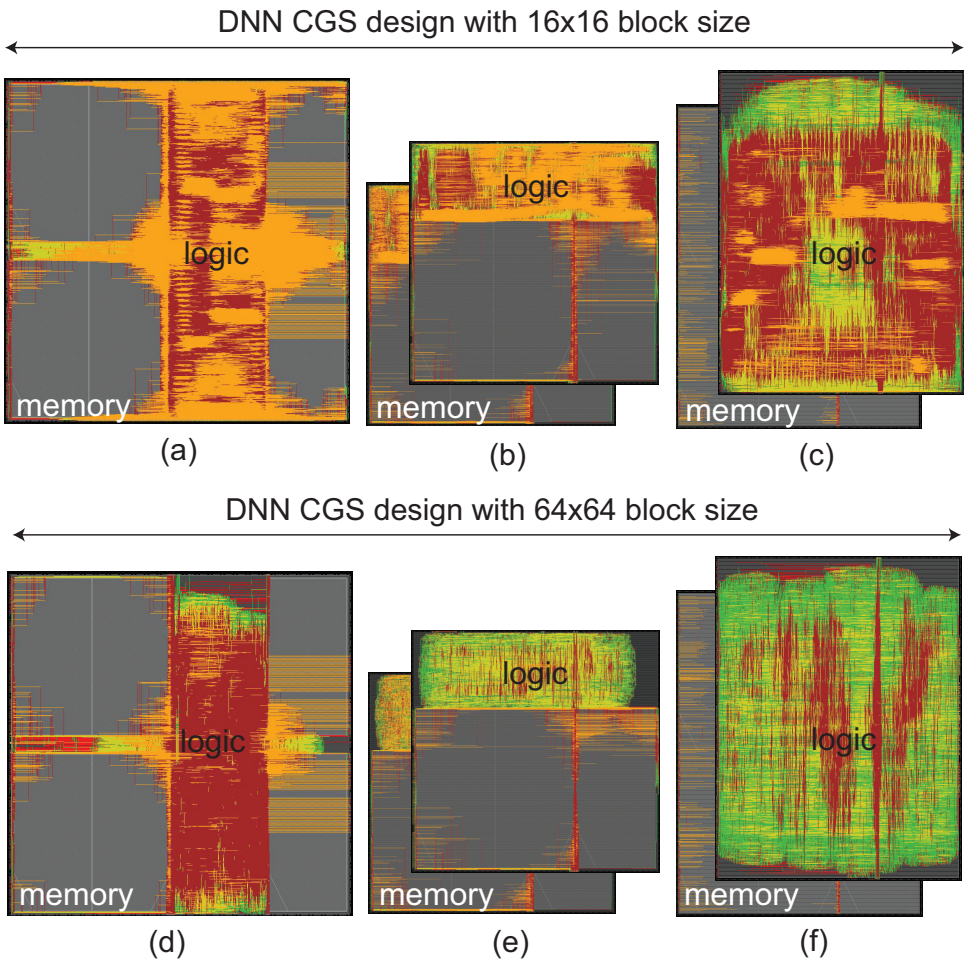



**Figure 4.5:** 28nm full-chip layouts of DNN CGS-16 and CGS-64 architectures at 400MHz target clock frequency. (a) 2D IC design, (b) M3D design with memory blocks on both tier (M3D-both), (c) M3D design with memory blocks on a single tier (M3D-one), (d) 2D IC, (e) M3D-both, (f) M3D-one.

## 4.5 M3D Impact on Energy-Efficiency

To analyze the advantage of M3D on energy-efficiency of different DNN architectures, two DNN architectures (DNN CGS-16 and CGS-64) are implemented using TSMC 28nm HPM technology with a target clock frequency of 400MHz. The footprint of 2D designs are set by targeting the initial standard cell density (excluding memory block area) before place-and-route to 65%. The impact of tier partitioning scheme is examined by comparing two memory floorplan schemes for M3D designs,

one with memory blocks on both tiers (M3D-both), and the other with memory blocks on a single tier only (M3D-one). In the M3D-both design, memory blocks are evenly split on the top and bottom tiers using similar floorplan for both tiers. On the other hand, in the M3D-one design, all standard cells are placed on one tier, and only memory blocks exist on the other tier. Fig. 4.5 shows the full-chip layouts of the implemented 2D and M3D designs.

### *4.5.1 Area, Wirelength, and Capacitance Comparisons*

Iso-performance comparison of several key metrics of the 2D and M3D designs is presented in Table 4.2. We summarize our findings as follows:

- **Footprint**: our M3D-both designs achieve 50.1% footprint reduction compared with the 2D designs, whereas the M3D-one designs obtain only 33.9% reduction. This difference is attributed to the large memory area compared with logic: $1.287mm^2$ vs. $0.505mm^2$ in the 2D CGS-16 design, for example. These large memory blocks, if placed in the same tier, cause the footprint to increase significantly.

- **Wirelength**: our wirelength saving reaches 29.9% and 33.7% in CGS-16 and CGS-64, respectively, with our M3D-both designs. This significant wirelength saving comes from 50% smaller footprint and shorter distance among cells in M3D designs.

- **Cell area**: we achieve 12.1% cell count reduction, which leads to 14.6% total cell area saving in our M3D-both design for CGS-16 architecture. This saving mainly comes from fewer buffers and smaller gates needed to close timing in M3D designs compared with the 2D counterparts. Our savings in CGS-64

architecture are 8.2% and 14.3% for the cell count and area, respectively.

- **MIV usage**: we use 77K MIVs in our CGS-16 architecture, while 48K MIVs are used in CGS-64. This is mainly because CGS-16 design is more complex than CGS-64 (to be further discussed in Section 4.7.1) so that our tier partitioning cutline cuts through more inter-tier connections in CGS-16. In the M3D-one design, logic and memory are separated into different tiers. This logic-memory connectivity is not high in our DNN architecture (= 1.7K).

- **Capacitance**: In our CGS-16 architecture, the 16.5% pin capacitance saving is from cell area reduction, while the 35.0% wire capacitance saving is from wirelength reduction. By comparing the raw data (943.3$pF$ vs. 2,216.8$pF$ in the 2D design), we note that our DNN architecture is wire-dominated. Our pin/wire capacitance saving reaches 25.0% and 37.7% in CGS-64.

To better understand why M3D-one gives significantly worse results than M3D-both, we show a placement comparison among 2D, M3D-both, and M3D-one designs in Fig. 4.6. In the M3D-both design shown in Fig. 4.6(b), the logic cells related to memory blocks in the top tier are placed in the same tier as the memory and densely packed to reduce wirelength effectively. This is the same for the bottom tier in the M3D-both design. On the other hand, we see that logic gates are rather spread out across the top tier in the M3D-one design shown in Fig. 4.6(c). This results in 1.1% *increase* in wirelength for CGS-16 and 26.7% *increase* in wirelength for CGS-64 compared with the 2D counterparts. This highlights the importance of footprint management and tier partitioning in the presence of large memory modules in DNN architectures.

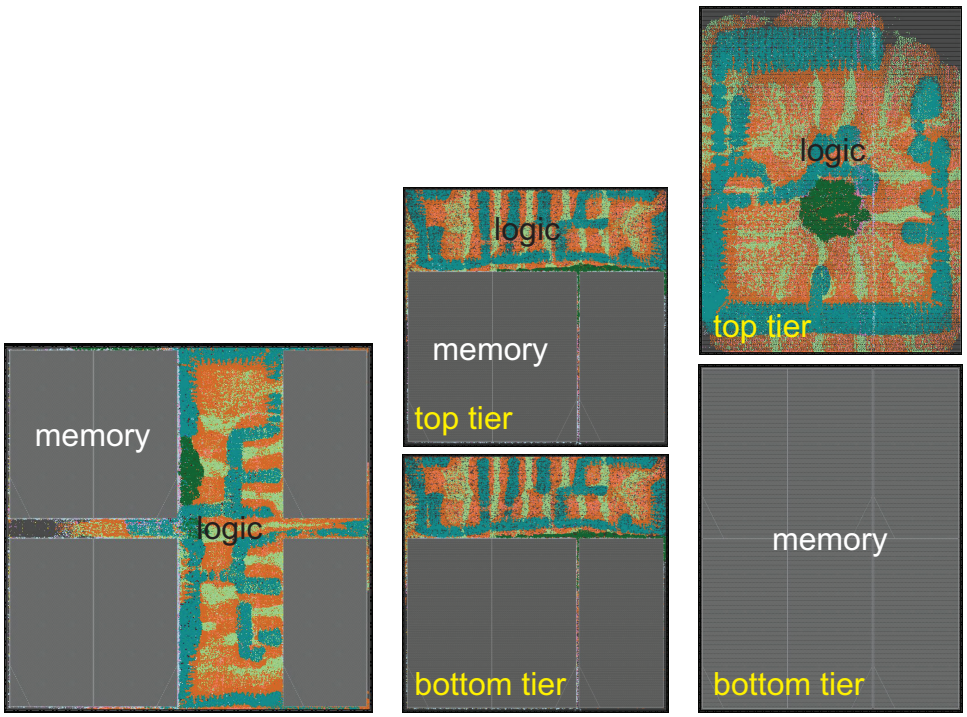



**Figure 4.6:** Cell placement of the modules in CGS-16 architecture. (a) 2D, (b) M3D-both, (c) M3D-one. Each module is highlighted with different colors.

### *4.5.2 Power Comparisons*

Table 4.3 presents the iso-performance power comparison between 2D and M3D designs of CGS-based DNNs. We report internal, switching, and leakage breakdown for each design. Our sign-off power calculations are conducted using two speech recognition workloads: classification and pseudo-training (more details provided in Section 4.7.2). From examining the power metrics of the 2D designs only, we observe the following:

- **CGS-16 vs. CGS-64**: during classification, CGS-16 consumes 141.1$mW$, while CGS-64 consumes 129.1$mW$. This confirms that CGS-16 consumes more power to handle more complex weight selection process (to be further discussed in Section 4.7.1). A similar trend is observed during pseudo-training: 220.0$mW$

vs. $176.3mW$.

- **Classification vs. pseudo-training**: pseudo-training, as expected, causes more switching in the circuits, and thus more power consumption compared with classification: $220.0mW$ vs. $141.1mW$ for CGS-16. A similar trend is observed for CGS-64: $176.3mW$ vs. $129.1mW$.

Next, we compare 2D vs. M3D power consumption. To explain the power reduction of M3D designs, Eq. (4.4) is employed, which describes the components comprising dynamic power consumption.

$$\begin{aligned} P_{dyn} &= P_{INT} + P_{SW} \\ &= \alpha_{IN} \cdot I_{SC} \cdot V_{DD} \cdot f_{clk} \\ &+ \alpha_{OUT} \cdot (C_{pin} + C_{wire}) \cdot {V_{DD}}^2 \cdot f_{clk} \end{aligned} \tag{4.4}$$

The first term $P_{INT}$ indicates the internal power consumption of standard cells and memory blocks. $P_{INT}$ is the product of short-circuit current ($I_{SC}$) during input switching, input activity factor $\alpha_{IN}$, clock frequency $f_{clk}$ and $V_{DD}$. The second term $P_{SW}$ represents the switching power dissipated during the charging or discharging of output load capacitance of cells ($C_{pin} + C_{wire}$). It is represented by the product of the output load capacitance, output activity factor $\alpha_{OUT}$, $f_{clk}$ and $V_{DD}$.

The resulting footprint of M3D-both designs is reduced by half, thereby reducing the wirelength between the cells. Fig. 4.7 shows the wirelength distribution of the 2D and M3D designs of CGS-16 architecture. The histogram clearly shows that M3D designs contain more number of short wires and fewer long wires compared with 2D. The effect of wirelength saving translates to the reduction of wire capacitance $C_{wire}$ in Eq. (4.4), therefore the saving of $P_{SW}$. Fig. 4.8 presents the distribution of standard cells with different ranges of cell drive-strength. We observe that M3D-both design

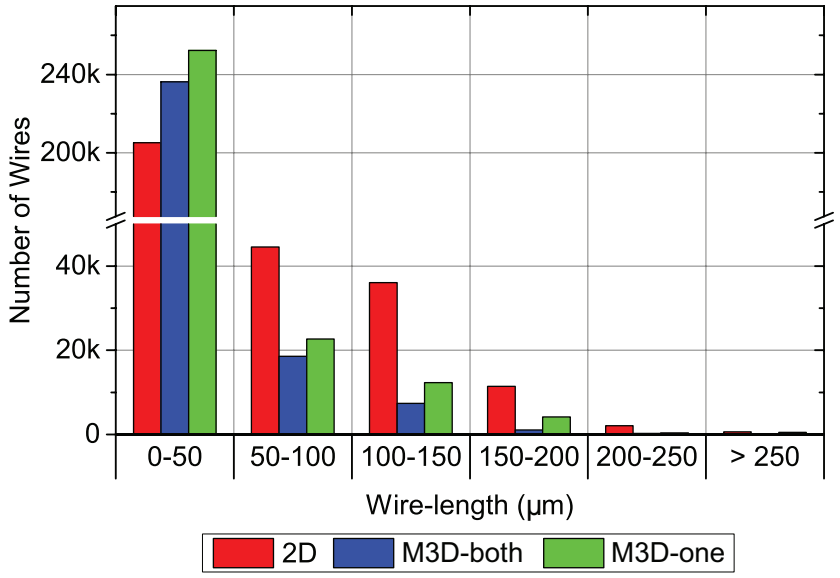



**Figure 4.7:** Wirelength distribution of CGS-16 architecture.

uses more number of low drive-strength cells (i.e. ×0-×0.8) and fewer high drive-strength cells (i.e. ×1-×16). Since low drive-strength cells utilize smaller transistors, their $I_{SC}$ and $C_{pin}$ are lower, which reduces both $P_{INT}$ and $P_{SW}$ in Eq. (4.4).

## 4.6 M3D Impact on Performance

In this section, we investigate the impact of M3D on performance of CGS-16 and CGS-64 architectures by pushing the target clock frequency of 2D and M3D designs to their maximum clock frequency. 2D and M3D designs are implemented with TSMC 28nm HPM technology sweeping the target frequency from 400MHz in 25MHz increments. The floorplans of the 2D and M3D designs are same as the ones used in Section 4.5. As M3D-both designs show better design quality compared to M3D-one designs as discussed in the section, we place memory blocks on both tiers in the M3D designs for this experiment as shown in Fig. 4.9.

The maximum performance comparison between the 2D and M3D designs of CGS-16 and CGS-64 architectures is presented in Table 4.4. The table shows the target

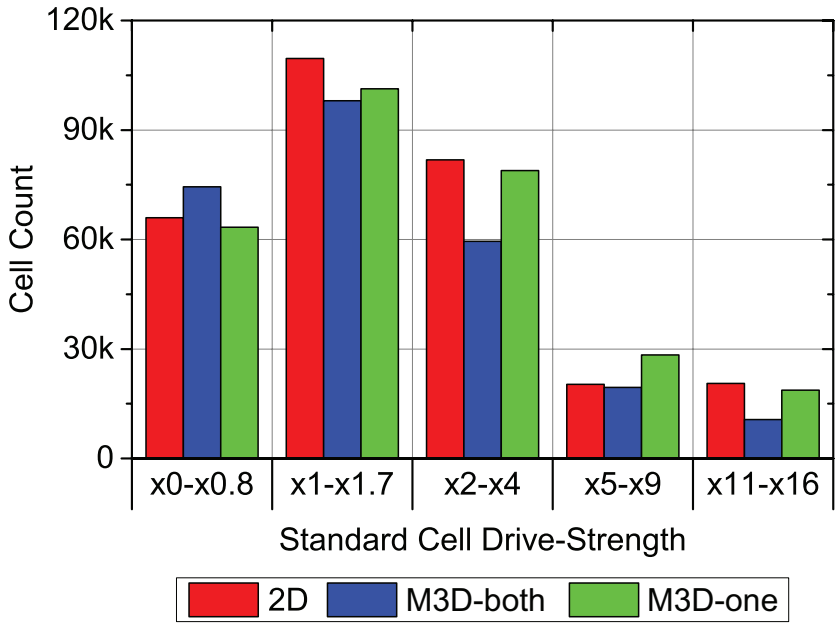



**Figure 4.8:** Cell drive-strength distribution of CGS-16 architecture.

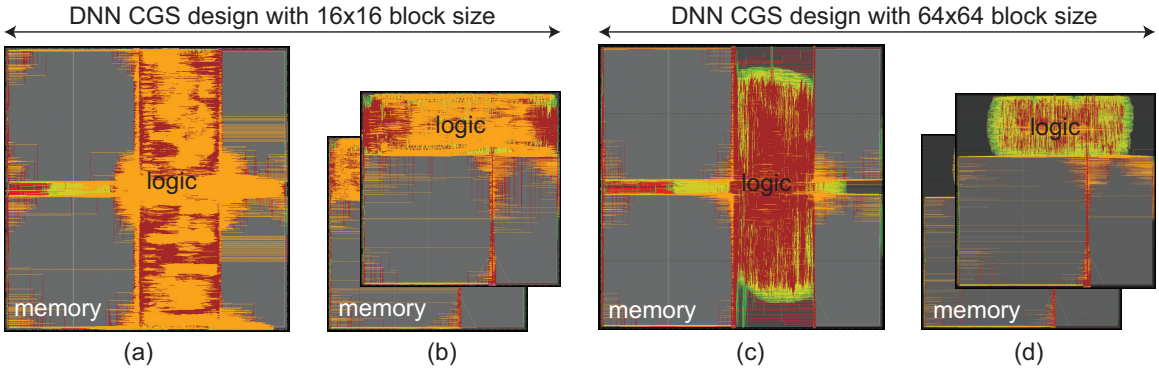



**Figure 4.9:** Full-chip die images of 2D and M3D designs of DNN CGS-16 and CGS-64 architectures at their maximum target clock frequency. (a) 2D design at 550MHz, (b) M3D design at 575MHz of DNN CGS-16 architecture, (c) 2D design at 600MHz, (d) M3D design at 625MHz of DNN CGS-64 architecture.

clock frequency used to place-and-route the designs, the resulting worst negative slack (WNS) from static timing analysis, and the effective clock frequency, which is the maximum achievable clock frequency that the designs are able to operate at without timing violation.

Comparing only the 2D designs of CGS-16 and CGS-64 architectures, we observe the following:

- **CGS-16 vs. CGS-64**: the effective clock frequency of the 2D CGS-16 design is 11.1% less than the 2D CGS-64 design. As the critical path of the 2D CGS-16 design starts from weight SRAM to MAC unit through weight selection logic, the lower effective clock frequency of the 2D CGS-16 design is attributed to its more complex weight selection logic as shown in a higher design density in Fig. 4.9(a) compared to Fig. 4.9(c).

Next, we compare the maximum performance of the 2D and M3D designs. Our M3D designs shows 6.2% and 1.2% performance improvement over 2D counterparts in CGS-16 and CGS-64 architectures, respectively. To analyze this trend, we first conduct the worst timing path comparison of the 2D and M3D designs. Fig. 4.10 compares the same timing path (i.e., the worst timing path of the 2D design) in the 2D and M3D CGS-16 designs at the maximum target clock frequency of the 2D design, and Table 4.5 presents key metrics of the timing path. The followings summarize our observations:

- **Wirelength**: the wirelength of the worst timing path of the 2D design is 53.6% longer than the same timing path in the M3D design. This is attributed to the reduced footprint and the inter-tier connections of the M3D design, which results in shorter distance among cells along the timing path.

- **Cell**: our M3D design offers 24.6% cell count saving as well as 21.3% average cell drive-strength reduction, thereby reducing cell area by 63.1% of the timing path. This is because fewer and smaller buffers are needed to drive the reduced wire load, which is a result of the wirelength reduction.

- **Capacitance**: compared to the 2D design, the wire and pin capacitance of the timing path in the M3D design are reduced by 51.6% and 35.8%, respectively. The wire capacitance reduction mainly comes from the wirelength reduction of

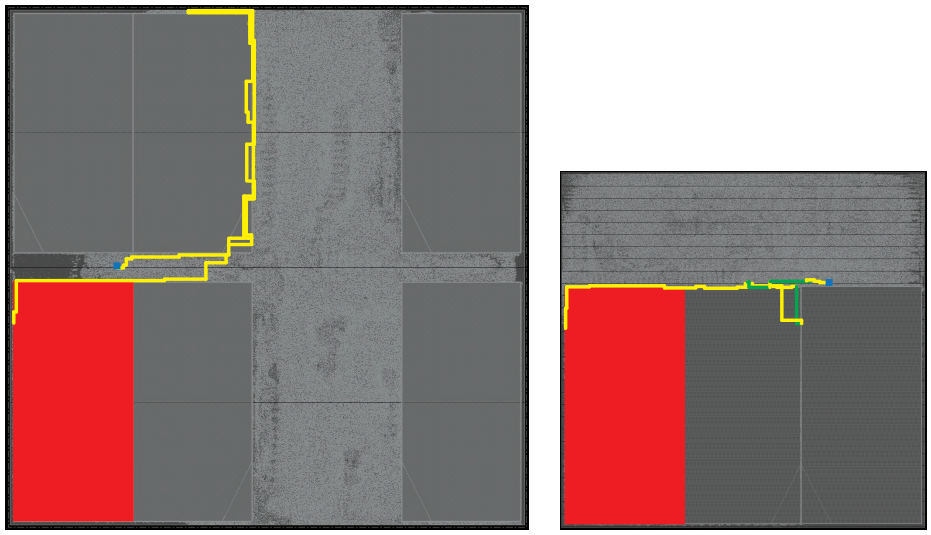



**Figure 4.10:** Worst timing path comparison of 2D and M3D designs of CGS-16 architecture. (a) The worst timing path of 2D design at its maximum target clock frequency, 550MHz. (b) The same timing path in M3D design. Cells in the top tier are projected into the bottom tier for M3D design, and red boxes (i.e., weight SRAM) indicate the start point, whereas blue boxes (i.e., flip-flops in MAC unit) represent the end point of the timing path. Yellow lines show the wires in 2D and the bottom tier of M3D design, whereas green lines are the top tier wires in M3D design.

the timing path, whereas the pin capacitance saving results from the cell count and cell drive-strength reduction.

- **Resistance**: our M3D design achieves 35.9% resistance reduction in the timing path. The resistance saving is also attributed to the wirelength saving along the timing path.

- **Delay**: due to the capacitance and resistance saving of the worst timing path, the delay of the timing path is reduced by 10.9% in the M3D design, thereby offering rooms to improve the performance.

In order to understand the impact of the above observations to the overall timing paths of the 2D and M3D designs, we report the slack distribution of all timing paths of the 2D and M3D CGS-16 designs in Fig. 4.11. While 18 timing paths of the

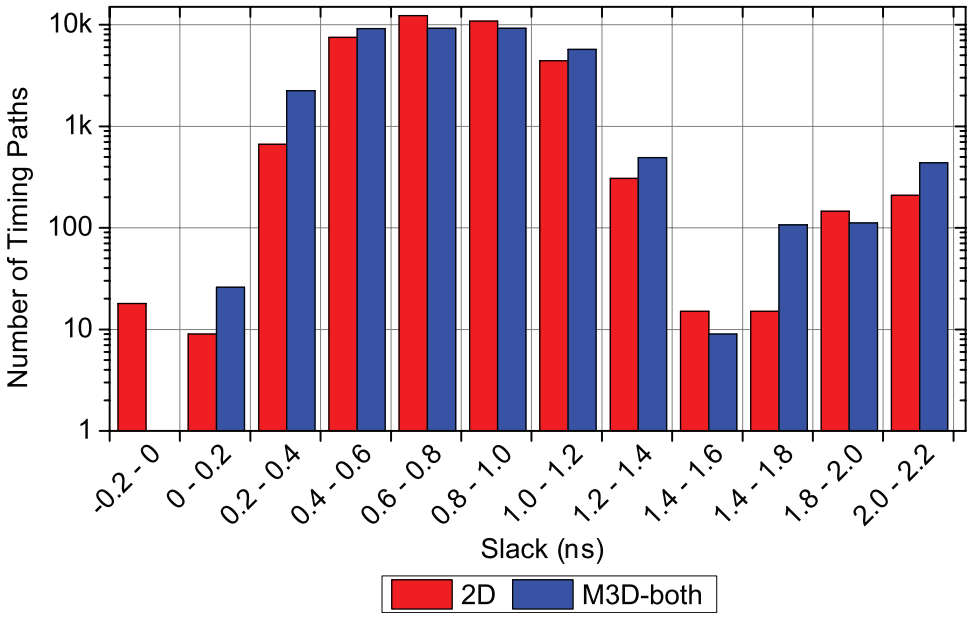



**Figure 4.11:** Slack distribution comparison between 2D and M3D designs of DNN CGS-16 architecture at the maximum clock frequency of the M3D design.

2D design violate the timing constraints, the M3D design successfully closes timing without any violation. In addition, we observe that there are more timing paths with high positive slack in the M3D design, which indicates that timing is easily closed in the M3D design due to the reduced delay of the timing paths.

The difference in the performance improvement of the M3D designs of CGS-16 and CGS-64 architecture is also attributed to the complex weight selection logic in CGS-16 and will be discussed in detail in Section 4.7.1

## 4.7 Architectural Impact Discussions

### *4.7.1 CGS-16 vs. CGS-64 Architecture Comparisons*

Table 4.3 shows that the total power reduction of M3D designs is higher in DNN CGS-16 architecture than CGS-64. Furthermore, we achieve more performance improvement with M3D in DNN CGS-16 architecture as shown in Table 4.4. These differences are caused by the granularity of weight selection methodology, i.e., coarse-

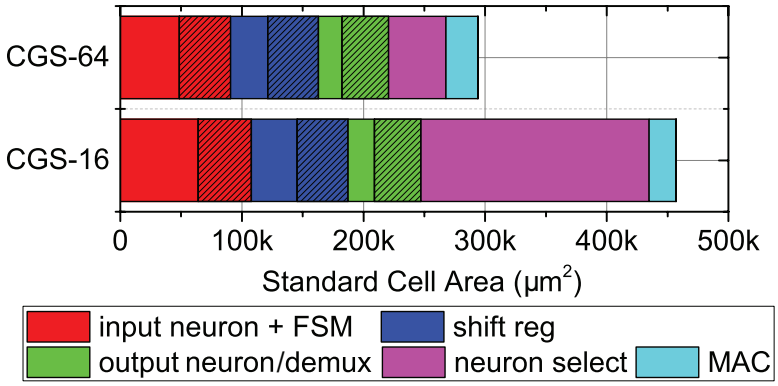



**Figure 4.12:** Standard cell area breakdown of 2D CGS-16 and 2D CGS-64 architectures. Non-dashed and dashed boxes respectively indicates combinational and sequential elements. Only five largest modules are shown.

grain sparsification (CGS) algorithm. The 1024×1024 weight matrix is divided into 256 (= 16×16) weight blocks in CGS-64 architecture. This count becomes 4,096 (= 64×64) weight blocks in CGS-16. The implication in DNN architecture is that CGS-16 requires a more complex neuron selection unit than CGS-64. Fig. 4.12 shows the comparison of standard cell area of each module in CGS-16 and CGS-64 architectures. We show both sequential (dashed box) and combinational logic (non-dashed box) portion in each module. We observe that the neuron selection unit in CGS-16 architecture (shown in purple) occupies more area than that in CGS-64 architecture.

As discussed in Section 4.5.1, M3D designs benefit not only from wirelength reduction but also from standard cell area saving. The number of storage elements (i.e. sequential logic and memory blocks) used in 2D and M3D designs remain the same. Thus, the only possible power reduction coming from storage elements is their drive strength reduction. This does not show a huge impact considering the small portion of sequential elements in our DNN architectures (16.1% on average). On the other hand, combinational logic can be optimized in various ways, such as logic reconstructing and buffer reduction. Therefore, our DNN M3D designs benefit more from combinational logic gates than sequential elements.

Fig. 4.13 shows the breakdown of total power consumption into combinational,

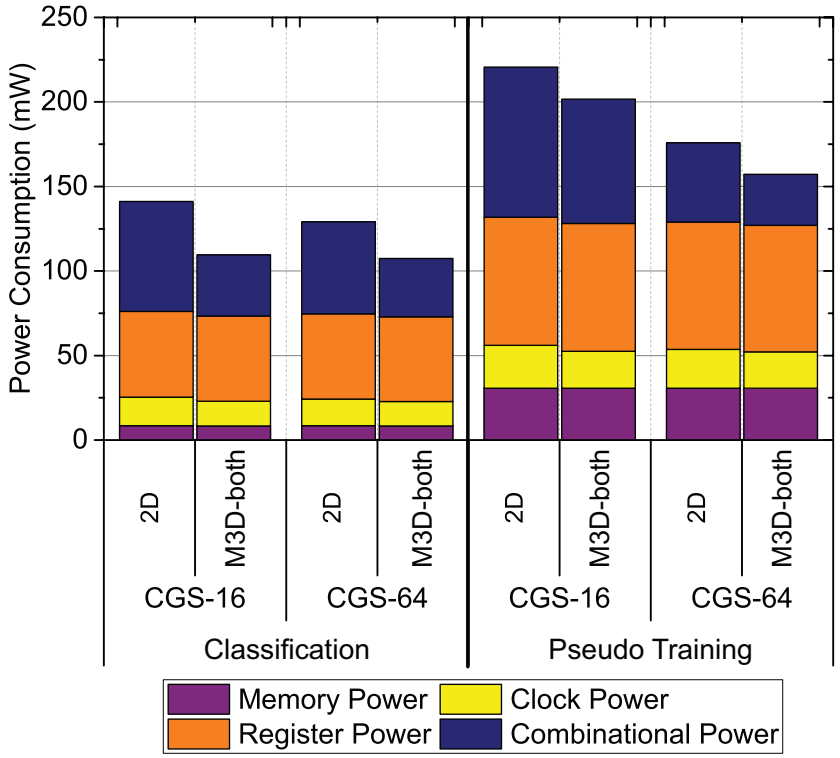



**Figure 4.13:** Power breakdown under two architectures (CGS-16 vs. CGS-64), two workloads (classification vs. pseudo-training), and two designs (2D vs. M3D).

register, clock, and memory portions. We see that combinational power reduction is the dominant factor in total power saving of M3D designs in both CGS-16 and CGS-64 architectures and in both classification and pseudo-training workloads. We also observe that the saving in other parts including register, clock, and memory power largely remain small. In addition, the neuron selection unit in CGS-16 architecture consists of a larger number of combinational logic gates than CGS-64. Thus, its M3D designs have more room for power optimization, resulting in a larger combinational power saving.

The larger neuron selection logic in CGS-16 architecture also offers more opportunity to improve the performance of M3D designs. While 2D designs suffer long timing path due to the complex neuron selection logic, M3D designs effectively reduce the wirelength, providing buffer count/size reduction along the worst timing path. This

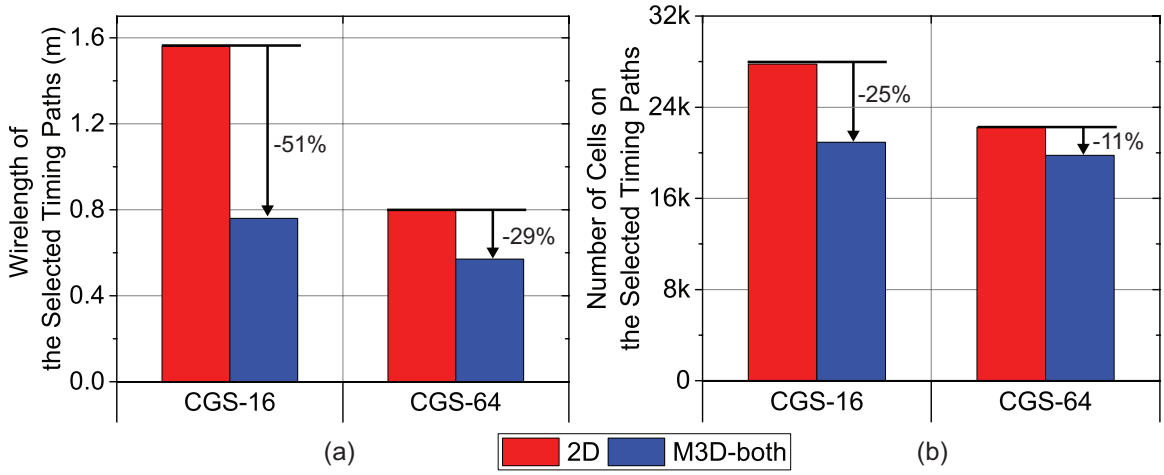



**Figure 4.14:** Comparison of (a) the total wirelength and (b) the total cell count of the timing paths from weight SRAMs to registers in MAC units through neuron selection logic of 2D and M3D-both designs of CGS-16 and CGS-64 architecture.

reduces the capacitance and resistance of timing paths, thereby offering shorter delay and larger performance improvement.

Fig. 4.14 compares the total wirelength and standard cell count along the selected 486 timing paths, which are from weight SRAMs to registers of MAC units through neuron selection logic, in the 2D/M3D CGS-16/CGS-64 designs at the maximum frequency of the 2D designs. Comparing only the 2D designs, the 2D CGS-16 design clearly utilizes longer wirelength as well as more standard cells as the neuron selection logic is more complex. As the M3D CGS-16 design has more combinational logics to optimize with the reduced footprint, it offers more cell count and wirelength reduction compared to the M3D CGS-64 design, providing more rooms for performance improvement in higher clock frequency.

### *4.7.2 Impact of Workloads*

In order to investigate the impact of different DNN workloads on M3D power reduction, we analyzed two main types of speech recognition DNN workloads: feed-forward classification and training. Real-world test vectors are used for feed-forward

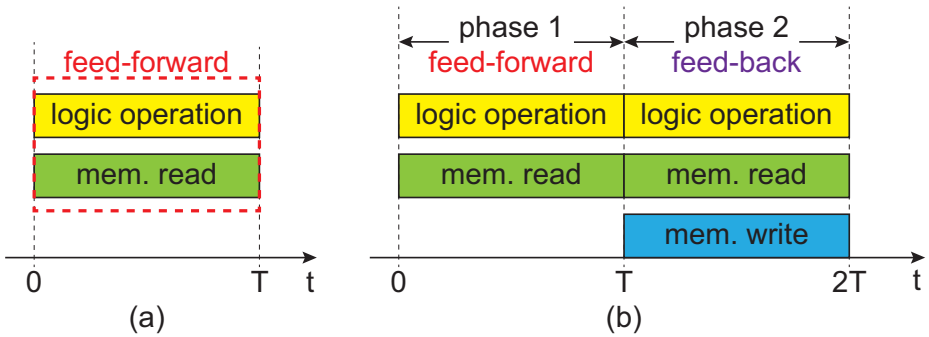



**Figure 4.15:** Comparison between (a) the feed-forward classification and (b) pseudo-training

classification. However, since our current architecture does not supports online training to avoid computational overhead of finding gradients in DNN training, we create customized test vectors for "pseudo-training". Online training on DNN consists of feed-forward computation and backward computation. In order to mimic the online training on the current architecture, there are two phases in our pseudo-training test vectors as shown in Fig. 4.15. In the first phase, the DNN performs feed-forward classification, which represents feed-forward computation during training. In the second phase, the DNN conducts feed-forward classification and writes the weights to memory blocks, which represents backward computation and weight update. These two phases mimic the behavior of logic computation and weight update during training.

Table 4.3 shows that while M3D-both shows 22.3% (CGS-16) and 16.9% (CGS-64) total power reduction in feed-forward classification workload, the power saving of pseudo-training workload is only 8.6% (CGS-16) and 10.7% (CGS-64). This difference stems from different switching patterns of combinational logic and storage elements in our DNN architecture. Our DNN mainly uses combinational logic gates to compute the values of neuron outputs and access memory for read operations only during feed-forward classification. Thus, this workload is classified as a compute-intensive kernel. On the other hand, memory operations are heavily used during pseudo-training since our DNN architecture needs to read and write weights. This

becomes a memory-intensive kernel. Therefore, switching activity in memory blocks is much higher during pseudo-training while that of combinational logic remains largely similar. This explains larger power consumption during pseudo-training workload: $220.0mW$ vs. $141.1mW$ for CGS-16, and $176.3mW$ vs. $129.1mW$ for CGS-64 as shown in Table 4.3.

As shown in Fig. 4.13, memory power and register power occupy a large portion of the total power during pseudo-training. This means that the combinational logic power saving becomes a smaller portion of the total power saving during training. The opposite is true for classification, where memory and register power are less dominant. In this case, the reduction in combinational power saving becomes more prominent in the total power saving.

## 4.8 Observations and Guidelines

We summarize the lessons learned from this study and provide design guidelines to maximize the power benefits of M3D designs targeting DNN architectures as follows.

- M3D effectively reduces the total power consumption of DNN architectures by reducing wirelength as well as standard cell area, showing its efficacy on saving power consumption of wire-dominated DNN circuits.

- M3D enhances the performance of DNN designs. This is mainly attributed to the reduced capacitance and resistance of timing paths in the designs, which comes from both wirelength and buffer count/size reduction.

- If memory blocks occupy more than half area of a DNN design, partitioning the memory blocks onto two tiers (i.e., M3D-both designs), instead of placing them on one tier (i.e., M3D-one designs), helps maximize the total power saving of the M3D design. It is because M3D-both designs achieve smaller footprint in that

case, which makes cell placement denser, and hence, reduces more wirelength.

- M3D shows larger power savings with smaller CGS block sizes, which consists of more combinational logics, in speech recognition DNNs. This enables the choice of selecting smaller block sizes for CGS in hardware implementations, which was earlier overlooked due to larger power overhead in 2D designs.

- DNNs with smaller CGS block sizes also benefit more on their performance from M3D, effectively reducing the overhead of more complex weight selection logic.

- In our DNN, it was combinational logic power reduction, not the commonly believed memory-related power reduction, that dominates the overall power saving of M3D. Moreover, compute-intensive classification workload gave us more power saving than memory-intensive training workload with M3D. Such a claim cannot become a general statement, and other DNN architectures may prove to be the opposite. However, we believe that the design and analysis methodologies presented in this paper pave a road for practical and convincing studies with other DNN architectures and their M3D implementations.

## 4.9 Conclusions

In this paper, we investigate the impact of M3D technology on power, performance, and area with speech recognition DNN architectures that exhibit coarse-grain sparsity. Our study shows that M3D reduces the total power consumption more effectively with compute-intensive workloads, compared to memory-intensive workloads. By placing memory blocks evenly on both tiers, M3D designs reduce the total power consumption up to 22.3%. This trend can be further extended to offer greater power reduction by using in-training quantization in conjunction with structured compression as demonstrated in Yin *et al.* (2018), and will be explored in future works. In

addition, owing to the reduced footprint and vertical integration, M3D designs offer performance improvement over 2D designs, especially in architecture with complex combinational logics. This study convincingly demonstrates the low power and high performance benefits of M3D on DNN hardware implementations and offers architectural guidelines to maximize the benefits.

**Table 4.2:** Iso-performance (400MHz) comparison of design metrics of 2D and M3D designs of DNN CGS-16 and DNN CGS-64 architectures. All percentage values show the reduction from their 2D counterparts.

| parameter | 2D | M3D-both | | M3D-one | |
|---|---|---|---|---|---|
| DNN CGS-16 | | | | | |
| footprint ($um$) | 1411×1411 | 1010×984 | **(-50.1 %)** | 996×1322 | (-33.9 %) |
| wirelength ($m$) | 12.089 | 8.469 | **(-29.9 %)** | 12.225 | (1.1 %) |
| cell count | 298,309 | 262,084 | **(-12.1 %)** | 290,692 | (-2.6 %) |
| cell area ($mm^2$) | 0.505 | 0.431 | **(-14.6 %)** | 0.511 | (1.1 %) |
| mem area ($mm^2$) | 1.287 | 1.287 | (0.0 %) | 1.287 | (0.0 %) |
| MIV count | - | 77,536 | | 1,776 | |
| pin cap ($pF$) | 943.3 | 788.0 | **(-16.5 %)** | 1,004.1 | (6.4 %) |
| wire cap ($pF$) | 2,216.8 | 1,440.8 | **(-35.0 %)** | 2,087.4 | (-5.8 %) |
| total cap ($pF$) | 3,160.1 | 2,228.7 | **(-29.5 %)** | 3,091.6 | (-2.2 %) |
| DNN CGS-64 | | | | | |
| footprint ($um$) | 1411×1411 | 1010×984 | **(-50.1 %)** | 996×1322 | (-33.9 %) |
| wirelength ($m$) | 5.631 | 3.734 | **(-33.7 %)** | 7.134 | (26.7 %) |
| cell count | 163,361 | 149,921 | **(-8.2 %)** | 174,292 | (6.7 %) |
| cell area ($mm^2$) | 0.314 | 0.269 | **(-14.3 %)** | 0.328 | (4.7 %) |
| mem area ($mm^2$) | 1.287 | 1.287 | (0.0 %) | 1.287 | (0.0 %) |
| MIV count | - | 48,636 | | 1,776 | |
| pin cap ($pF$) | 520.8 | 390.8 | **(-25.0 %)** | 553.5 | (6.3 %) |
| wire cap ($pF$) | 920.1 | 573.7 | **(-37.7 %)** | 1,110.5 | (20.7 %) |
| total cap ($pF$) | 1,440.9 | 964.4 | **(-33.1 %)** | 1,664.0 | (15.5 %) |

**Table 4.3:** Iso-performance (400MHz) power comparison of two architectures (CGS-16 vs. CGS-64) using two workloads (classification vs. pseudo-training). All percentage values show the reduction from their 2D counterparts.

| workload | power breakdown | 2D | M3D-both | | M3D-one | |
|---|---|---|---|---|---|---|
| DNN CGS-16 | | | | | | |
| 4*classification | internal power ($mW$) | 91.3 | 76.7 | (-16.0 %) | 90.3 | (-1.1 %) |
| | switching power ($mW$) | 48.6 | 31.6 | (-35.0 %) | 46.5 | (-4.3 %) |
| | leakage power ($mW$) | 1.3 | 1.2 | (-6.6 %) | 1.3 | (0.5 %) |
| | total power ($mW$) | **141.1** | 109.6 | **(-22.3 %)** | 138.0 | (-2.2 %) |
| 4*pseudo-training | internal power ($mW$) | 150.4 | 142.8 | (-5.1 %) | 148.3 | (-1.4 %) |
| | switching power ($mW$) | 68.4 | 57.1 | (-16.6 %) | 65.6 | (-4.2 %) |
| | leakage power ($mW$) | 1.3 | 1.2 | (-6.8 %) | 1.3 | (0.7 %) |
| | total power ($mW$) | **220.0** | 201.0 | **(-8.6 %)** | 215.0 | (-2.3 %) |
| DNN CGS-64 | | | | | | |
| 4*classification | internal power ($mW$) | 86.8 | 76.1 | (-12.3 %) | 84.9 | (-2.2 %) |
| | switching power ($mW$) | 41.2 | 30.2 | (-26.7 %) | 42.8 | (3.9 %) |
| | leakage power ($mW$) | 1.1 | 1.1 | (-4.7 %) | 1.1 | (1.5 %) |
| | total power ($mW$) | **129.1** | 107.3 | **(-16.9 %)** | 128.8 | (-0.2 %) |
| 4*pseudo-training | internal power ($mW$) | 129.2 | 120.0 | (-7.2 %) | 128.5 | (-0.5 %) |
| | switching power ($mW$) | 46.0 | 36.3 | (-21.2 %) | 50.3 | (9.3 %) |
| | leakage power ($mW$) | 1.1 | 1.1 | (-4.6 %) | 1.1 | (1.4 %) |
| | total power ($mW$) | **176.3** | 157.4 | **(-10.7 %)** | 179.9 | (2.0 %) |

**Table 4.4:** Maximum performance comparison of 2D and M3D designs of CGS-16 and CGS-64 architectures.

| parameter | | DNN CGS-16 | DNN CGS-64 |
|---|---|---|---|
| 3*2D | target clk freq (MHz) | 550 | 600 |
| | WNS (ns) | -0.056 | 0.002 |
| | **effective clk freq (MHz)** | **534** | **601** |
| 3*M3D | target clk freq (MHz) | 575 | 625 |
| | WNS (ns) | -0.024 | -0.046 |
| | **effective clk freq (MHz)** | **567** | **608** |
| **$\Delta$% effective clk freq** | | **6.2%** | **1.2%** |

**Table 4.5:** Key parameter comparison of the worst timing path in Fig. 4.10 of the 2D and M3D designs of DNN CGS-16 architecture.

| parameter | 2D | M3D | |
|---|---|---|---|
| wirelength ($um$) | 3,208 | 1,488 | (-53.6%) |
| cell count | 65 | 49 | (-24.6%) |
| avg. cell drv-str | 8.9 | 7.0 | (-21.3%) |
| cell area | 180.1 | 66.4 | (-63.1%) |
| MIV count | - | 6 | |
| wire cap ($fF$) | 500 | 242 | (-51.6%) |
| pin cap ($fF$) | 486 | 312 | (-35.8%) |
| resistance ($k\Omega$) | 14.5 | 9.3 | (-35.9%) |
| **delay (ns)** | **2.344** | **2.088** | **(-10.9%)** |

Chapter 5

# HCGS: COMPRESSING LSTM NETWORKS WITH HIERARCHICAL COARSE-GRAIN SPARSITY

The long short-term memory (LSTM) network is one of the most widely used recurrent neural networks (RNNs) for automatic speech recognition (ASR), but requires millions of parameters. For memory constrained hardware accelerators, the increased demand for storage causes higher dependence on off-chip memory, resulting in throttled inference speed and higher power consumption. In this paper, we propose a new LSTM training technique based on hierarchical coarse-grain sparsity (HCGS), which enforces hierarchical structured sparsity by randomly dropping static blockwise connections between layers. HCGS maintains the same hierarchical structured sparsity throughout training and inference, which can aid acceleration and storage reduction for both training and inference hardware systems. We jointly investigate HCGS-based structured sparsity and in-training quantization on 2-/3-layer LSTM networks for TIMIT and TED-LIUM corpora. With 16× structured compression and 6-bit weight precision, we achieved a phoneme error rate (PER) of 16.9% for TIMIT and a word error rate (WER) of 18.9% for TED-LIUM, showing the best trade-off between error rate and LSTM memory compared to prior compression works.

## 5.1 Introduction

The advent of internet of things (IoT) and edge computing has created a requirement of energy-efficient computation of deep neural networks (DNNs) on mobile devices Shafique *et al.* (2018).

The particular challenge of performing on-device Automatic Speech Recognition

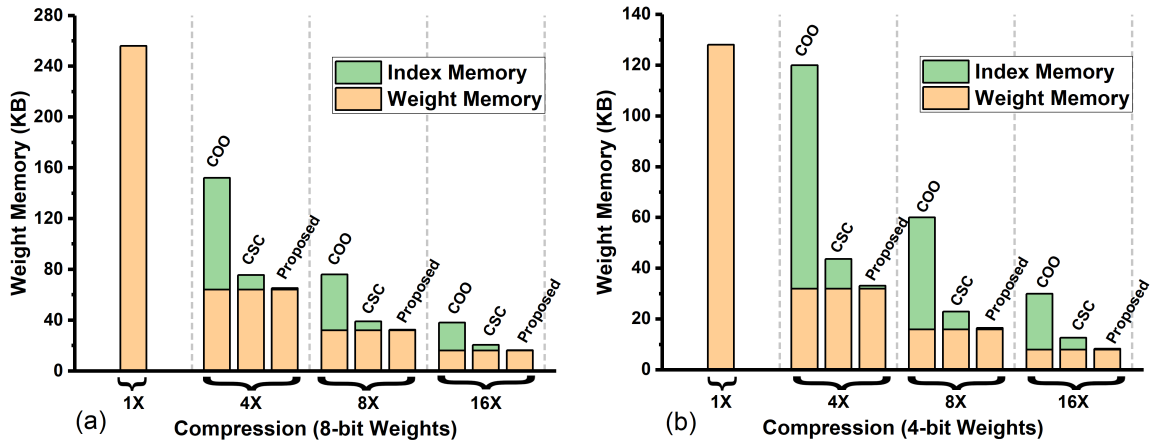



**Figure 5.1:** Comparison of index and weight memory requirement among three compression methodologies for 4×, 8×, and 16× compression. (a) Compression of 8-bit weights. (b) Compression of 4-bit weights.

(ASR) is that state-of-the-art LSTM based models for ASR contain tens of millions of weights Xiong *et al.* (2018); Han *et al.* (2018). Weights can be stored on-chip (e.g. SRAM cache of mobile processors), which has fast access time (nanoseconds range), but is limited to a few mega bytes (MB) due to cost Halpern *et al.* (2016). Alternatively, weights can be stored off-chip (e.g. DRAM) up to a few gigabytes (GB), but access is slower (tens of nanoseconds range) and consumes ∼100× higher energy than on-chip counterparts Han *et al.* (2015b). To improve energy-efficiency of DNN hardware, off-chip memory access and communication need to be minimized Chen *et al.* (2017). To that end, it becomes crucial to store most or all weights on-chip through sparsity/compression, weight quantization, and network size reduction.

DNN model compression has been heavily investigated in the literature Han *et al.* (2015a); Cheng *et al.* (2015); Tu *et al.* (2016); Kadetotad *et al.* (2016); Wang *et al.* (2018); Louizos *et al.* (2017); Zhu and Gupta (2017); Wen *et al.* (2016); Zhu *et al.* (2018). Element-wise sparsity can result in a large compression of DNN weights Han *et al.* (2015a); Narang *et al.* (2017), but the index storage can be as large as the non-zero weights themselves, especially if we use the simple coordinate (COO) format

that stores the location of each non-zero weight. The compressed sparse row (CSR) or compressed sparse column (CSC) format Han *et al.* (2017) reduces the index cost as only the distance between non-zero elements in a row/column is stored, but still exhibits noticeable index memory and causes irregular memory access Wang *et al.* (2018). Moreover, several recent works have investigated joint optimization of compression and low-precision quantization Yin *et al.* (2017c); Ye *et al.* (2018), and for weights with lower precision, the relative cost of index storage will be even higher. To share the index for a large number of weights, row-/column-/block-wise structured or coarse-grain compression have been proposed Kadetotad *et al.* (2016); Wen *et al.* (2016); Yu *et al.* (2017); Wen *et al.* (2017); Wang *et al.* (2018), which minimizes the index storage, makes memory access more regular, and enhances DNN inference acceleration.

Using the three aforementioned compression methods of COO, CSC, and structured sparsity, Figure 5.1 shows the comparison of index memory overhead for compression targets from 1× (dense network) to 16× (only 6.25% of weights are non-zero), considering a 512×512 weight matrix with 8-bit and 4-bit precision per weight. Compared to 8-bit weights (Figure 5.1(a)), the index overhead roughly doubles with 4-bit weights (Figure 5.1(b)).

In this work, we introduce hierarchical block-wise sparsity for weights matrices in LSTMs, which substantially reduces the index overhead to <1.3%, as shown in Figure 5.1.

For LSTM based RNNs, obtaining structured sparsity can be more challenging when compared to multi-layer perceptrons (MLPs) or convolutional neural networks (CNNs) due to the temporal dependency of the recurrent units. In addition, attempts to aggressively compress the RNN without taking into consideration the interconnected gates in the LSTM will lead to a mismatch in the dimension and adversely

affect the accuracy Wen *et al.* (2017). To minimize accuracy loss with large compression rates, we propose hierarchical coarse-grain sparsity (HCGS) for LSTMs. We enforce a hierarchically sparse structure between LSTM layers before training, by randomly selecting large blocks and then randomly selecting small blocks within the selected large blocks. This type of hierarchical block-wise sparsity is maintained statically throughout training and inference, which can aid acceleration and storage reduction not only for inference, but also potentially for training hardware systems.

We jointly investigated HCGS-based structured sparsity and in-training quantization Hubara *et al.* (2017) on 2-/3-layer LSTM RNNs for TIMIT Garofolo *et al.* (1990) and TED-LIUM Rousseau *et al.* (2012) corpora. With 16× structured compression and 6-bit weight precision, we achieved 16.9% PER for TIMIT with 0.3 MB of total RNN weight memory, and 18.9% WER for TED-LIUM with 1 MB of RNN weight memory. By evaluating various compression and quantization values on different sizes of LSTM RNNs, we determine the Pareto-optimal designs, where HCGS-based LSTMs show the best trade-off between error rate and weight memory compared to prior LSTM compression works.

Code implementing the models in this work is available at `https://github.com/razor1179/pytorch-kaldi-CGS`.

## 5.2 Related Work

The very large model size of LSTM RNNs prompted many prior works to investigate sparsity/compression techniques for deployment on mobile devices. In Narang *et al.* (2017), the pruning threshold is monotonically increased during training, and the final threshold is optimized to minimize accuracy loss and maximize compression. Using this technique, the size of Deep Speech 2 model Amodei *et al.* (2016) can be compressed from 268MB to ∼32MB. Sparse persistent RNN Zhu *et al.* (2018) presents

several optimizations for sparse RNNs including Lamport timestamps, wide memory loads, and a bank-aware weight layout, enhancing RNN acceleration on GPUs. ESE Han *et al.* (2017) prunes LSTM parameters that are smaller than chosen thresholds while minimizing accuracy loss, and the non-zero weights are condensed using CSC format. However, the sparse weight matrices obtained by element-wise pruning techniques have irregular memory access patterns. The resulting non-structured and fine-grained sparsity leads to the design of a complex sparse matrix multiplication module and results in limited speedup for RNN inference.

To overcome the inefficiency of element-wise sparsity, several prior works compressed DNNs with structured or coarse-grain sparsity, minimizing the index, regularizing memory access, and accelerating inference. Structured sparsity learning (SSL) Wen *et al.* (2016) has proposed row-/column-/layer-wise structured sparsity based on group Lasso regularization, leading to enhanced acceleration. But SSL was only demonstrated on CNNs and did not consider any quantization.

Block-wise sparsity was presented in Kadetotad *et al.* (2016), where static sparsity is applied on randomly selected blocks of weights during training. However, only simple MLPs have been demonstrated in this work and only 4× compression is achieved with a single-level block structure. For RNNs, structured sparsity techniques have been proposed with row-/column-wise sparsity in Wen *et al.* (2017), but the compression has been limited to 3× for iso-accuracy.

Employing block-circulant weight matrices for LSTMs Wang *et al.* (2018); Li *et al.* (2018) enables matrix multiplications to be computed with Fast Fourier transform (FFT) operations, reducing the LSTM model size and computational complexity. However, this advantage is lost if the block size is too small, which constrains the granularity of the resulting sparsity.

While most compression works perform rigorous training with all weights and

prune out the weights gradually or iteratively, our work only trains the compressed weights (randomly pre-determined blocks) and maintains the same block structure for inference as well. We first constrain the LSTM network with sparsity and rely on training process for the LSTM to adapt its weights to the enforced sparse structure. The hierarchically sparse structure (sparse small blocks within sparse large blocks) provides a good balance between coarse and fine connection granularity for high levels of compression.

## 5.3 HCGS Based LSTM Training

### *5.3.1 Long Short-Term Memory RNN*

RNNs construct a continuous and consecutive sequence of hidden states/representations $\{h_1, h_2, ..., h_T\}$ by processing corresponding input sequence $\{x_1, x_2, ..., x_T\}$. In single-layer RNNs, the hidden states $\{h_1, h_2, ..., h_T\}$ are used for prediction or decision making. In deep (stacked) RNNs, only the hidden states of the final layer are used for predictions while the hidden states in other layers are used as inputs to their corresponding next layers.

Each hidden state is implicitly trained to remember and emphasize task-relevant aspects of the preceding inputs, and is incorporated with new inputs via a recurrent operator, $T$. This operation converts the previous hidden state and the present input into a new hidden state, e.g.,

$$h_t = T(h_{t-1}, x_t) = tanh(Wx_t + Uh_{t-1} + b),$$

where $W$ and $U$ are weights for the feed-forward and recurrent structures, respectively, and $b$ is the bias parameter.

Figure 5.2 shows the computation flow for each layer of an LSTM Hochreiter and Schmidhuber (1997), which is a specialized recurrent structure. In addition to

the hidden state $h_t$ used as a transient representation of state at timestep $t$, LSTM introduces a memory cell $c_t$, intended for internal long-term storage. The parameters $c_t$ and $h_t$ are computed via input, output, and forget gate functions. The forget gate function $f_t$ directly connects $c_t$ to the memory cell $c_{t-1}$ of the previous timestep via an element-wise multiplication. Large values of the forget gates cause the cell to remember most (if not all) of its previous values. Each gate function has a weight matrix and a bias vector; we use subscripts $i$, $o$ and $f$ to denote parameters for the input, output and forget gate functions, respectively, e.g., the parameters for the forget gate function are denoted by $W_f$, $U_f$, and $b_f$.

With the above notations, an LSTM is defined as:

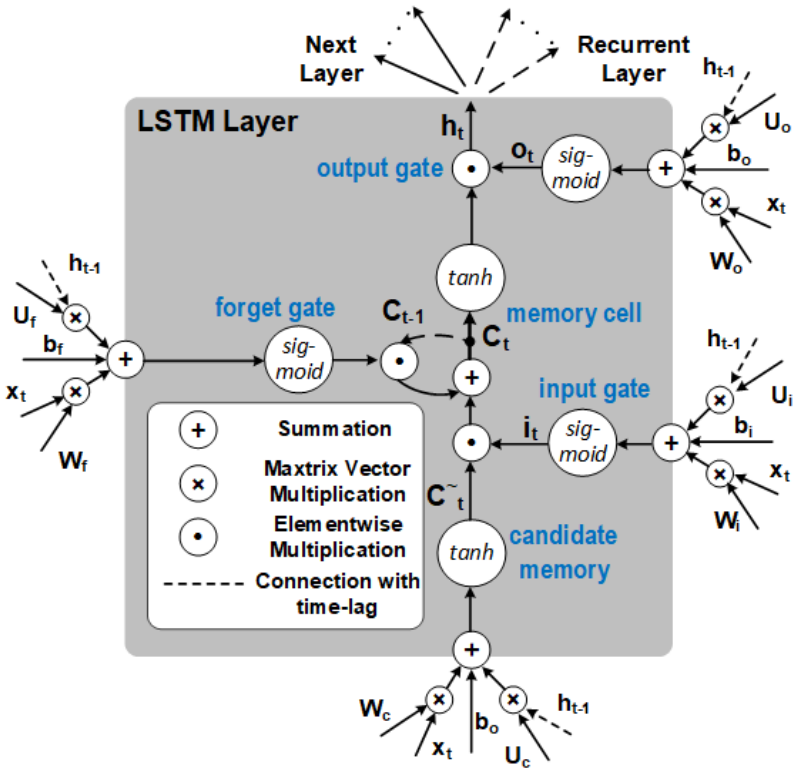



**Figure 5.2:** LSTM computation flow for each layer. Each of the four gates of the LSTM layer receives the input sequence $x_t$ and the recurrent hidden state sequence $h_{t-1}$, along with the corresponding weights and biases.

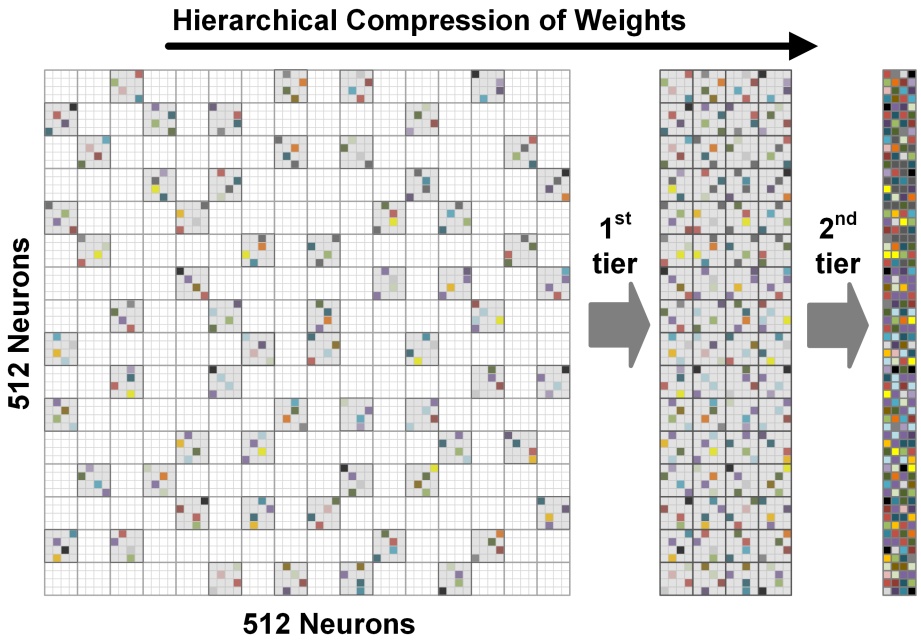



**Figure 5.3:** Proposed hierarchical block-wise compression of weights.

0.6em

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i), \tag{5.1}$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f), \tag{5.2}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o), \tag{5.3}$$

$$\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c), \tag{5.4}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \tag{5.5}$$

$$h_t = o_t \odot tanh(c_t), \tag{5.6}$$

where $\sigma(\cdot)$ represents the sigmoid function and $\odot$ is the element-wise product.

### *5.3.2 Hierarchical Coarse-Grain Sparsity (HCGS)*

We propose hierarchical coarse-grain sparsity (HCGS) for LSTM training, to introduce block-wise sparsity in LSTM weights. Compressed HCGS networks require

storage of only the non-zero weights and the corresponding indices for blocks. Figure 5.3 illustrates HCGS implementation for RNNs, where the connections between feed-forward layers and recurrent layers within the RNN are dropped in a hierarchical block-wise manner. The final compressed weight matrix in Figure 5.3 is obtained by including a binary connection mask ($C^W$ or $C^U$ in Algorithm 1) during the feed-forward computation of the LSTM. The binary connection mask is randomly initialized at the start of training and remains constant throughout the training process. It contains only $0's$ and $1's$, where $0's$ signify the deleted connections. The preserved connections are randomly selected in a hierarchical manner. The example shown in Figure 5.3 has a two-tier hierarchy to obtain the final sparsity. The first tier connections are dropped randomly in large blocks (i.e. grey blocks in Figure 5.3). Out of the preserved connections in the first tier (grey blocks), the second tier connections are then dropped randomly in smaller blocks to achieve the target sparsity. Only the weights corresponding to the preserved connections for both tiers of hierarchy are compressed and stored.

The indices needed for decompressing the weight matrix also have two tiers. The two-tier index has a recursive structure, where the index stores the location of a smaller block within the larger block. The index for a smaller weight block in Figure 5.3 consists of two tiers. The first tier stores the location of the grey block within the weight matrix, and the second tier represents the block's location within the larger grey block. It is important to note that all the smaller weight blocks within a larger grey block have the identical first tier of the index, so that the first tier of the index can be shared for smaller blocks that are within the same larger grey block. This further reduces the overall index memory requirement. This work focuses on two tiers of hierarchy for HCGS for design simplicity, but the hierarchy can be expanded to exhibit multiple tiers of block-wise sparse structure, recursively selecting

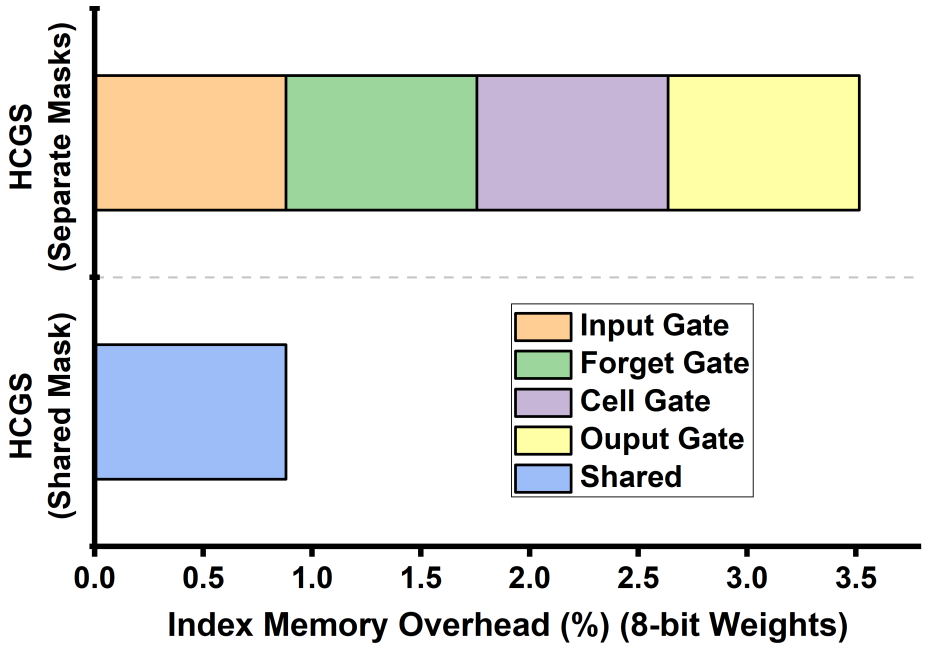



**Figure 5.4:** Further reduction of index memory aided by sharing the random connection mask for four gates in each LSTM layer.

even smaller blocks within smaller blocks.

Algorithm 1 shows the computational changes required to incorporate HCGS in LSTM training prior to using (1)-(5). The binary connection mask is initialized for every layer of the feed-forward network ($C^W$) and the recurrent network ($C^U$), which forces the deleted weight connections to zero. During back-propagation, the HCGS mask ensures that the deleted weights that have been forced to zero do not get updated and remain zero throughout training.

To further increase compression efficiency, weights associated with the four gates in each LSTM layer share the common connection maskthat is randomly selected. As shown in Figure 5.4, sharing the same random mask results in 4× reduction of the index memory, and reduces the computations for decompression by 4× as well. Compared to cases of using different random masks, sharing the same random mask for four gates did not affect PER or WER by more than 0.2% across all our LSTM experiments. It is important to note that the block sizes chosen in both tiers affect

the final accuracy of the trained network. Therefore, LSTM networks with varying block sizes in both tiers must be evaluated to obtain the optimal compression and accuracy. This will be discussed in Section **??**.

### *5.3.3 Quantizing LSTM Networks*

Quantization of parameters has traditionally been done post training of the DNN Han *et al.* (2017); Kadetotad *et al.* (2016). For very low-precision quantization, this approach can lead to accuracy degradation for quantized DNNs. A solution to achieve high accuracy with very low-precision quantization was proposed in Hubara *et al.* (2017), where weights of the DNN were quantized during training. Similar in-training quantization schemes have been employed for our quantization of LSTMs, to jointly optimize structure sparsity and low-precision quantization.

During the feed-forward part of the LSTM training, each weight is quantized to $n$ bits, while the feed-backward part uses full-precision weights. This way, the network is optimized to minimize the cost function with $n$-bit precision weights. The $n$-bit quantized weights are represented in (7) and steps to make quantized copies of the full-precision weights are shown in Algorithm 2.

$$W^{q_n} = Quantization(W, n) \tag{5.7}$$

To reflect quantization of the weights, the inference equations for the LSTM are altered as shown in (8)-(11):

0.6em

$$i_t = \sigma(BN(W_i^{q_n} x_t) + U_i^{q_n} h_{t-1}), \tag{5.8}$$

$$f_t = \sigma(BN(W_f^{q_n} x_t) + U_f^{q_n} h_{t-1}), \tag{5.9}$$

$$o_t = \sigma(BN(W_o^{q_n} x_t) + U_o^{q_n} h_{t-1}), \tag{5.10}$$

$$\tilde{c}_t = tanh(BN(W_c^{q_n} x_t) + U_c^{q_n} h_{t-1}), \tag{5.11}$$

where $BN(\cdot)$ refers to batch normalization Ioffe and Szegedy (2015), and the bias value is replaced by $\beta$ in batch normalization. Also, batch normalization was adopted only for feed-forward connections, as proposed in Laurent *et al.* (2016); Ravanelli *et al.* (2017).

In addition, the parameter update section in Algorithm 1 is altered to include the process of updating the batch normalization parameters. Back-propagation through time (BPTT) Werbos (1990) is used compute the gradients, where the gradients are calculated by minimizing the cost function using the quantized weights $W^{q_n}$, but the full-precision weight copies ($W$) are updated to ensure the network is optimized to reduce the output error for quantized weights.

## 5.4 Experiments

### *5.4.1 Experimental Setup*

For speech recognition applications considered here, we employ 440 fMLLR input features Gales *et al.* (1998), which are extracted using the s5 recipe of Kaldi Povey *et al.* (2011). The fMLLR features were computed using time windows of 25ms with an overlap of 10ms. We use the PyTorch-Kaldi speech recognition toolkit Ravanelli *et al.* (2018) to train the LSTM networks.

The final LSTM layer generates the acoustic posterior probabilities, which are

**Table 5.1:** Comparison of different LSTM network configurations that lead to the same memory storage requirements.

| Cells | Compression | No. Parameters | Bits per weight | Memory (MB) |
|---|---|---|---|---|
| 256 | 1× | 524,288 | 32 | 2 |
| 512 | 4× | 524,288 | 32 | 2 |
| 1,024 | 16× | 524,288 | 32 | 2 |
| 512 | 2× | 1,048,576 | 16 | 2 |
| 1,024 | 4× | 2,097,152 | 8 | 2 |
| 512 | 1× | 2,097,152 | 8 | 2 |
| 1,024 | 2× | 4,194,304 | 4 | 2 |
| 1,024 | 1× | 8,388,608 | 2 | 2 |

normalized by their prior before feeding them to a hidden Markov model (HMM) based decoder. An $n$-gram language model derived from the language probabilities is merged with the acoustic scores by the decoder. A beam search algorithm is then used to retrieve the sequence of words uttered in the speech signal. The final error rates for TIMIT and TED-LIUM corpora are computed with the NIST SCTK scoring toolkit SCTK (2008).

For the TIMIT corpus, we considered the phoneme recognition task (aligned with the Kaldi s5 recipe). We trained 2-layer uni-directional LSTM networks, with 256, 512, and 1,024 cells per layer. For the TED-LIUM corpus, we targeted the word recognition task (aligned with the Kaldi s5 recipe). We trained 3-layer uni-directional LSTM networks, with 256, 512, and 1,024 cells per layer.

Table 5.1 shows a few possible LSTM configurations of compression and quantization, which results in the same storage requirements for a LSTM layer. With the same storage or on-chip memory requirement, we should choose the network configuration that provides the best accuracy. To characterize the accuracy-memory trade-off, a

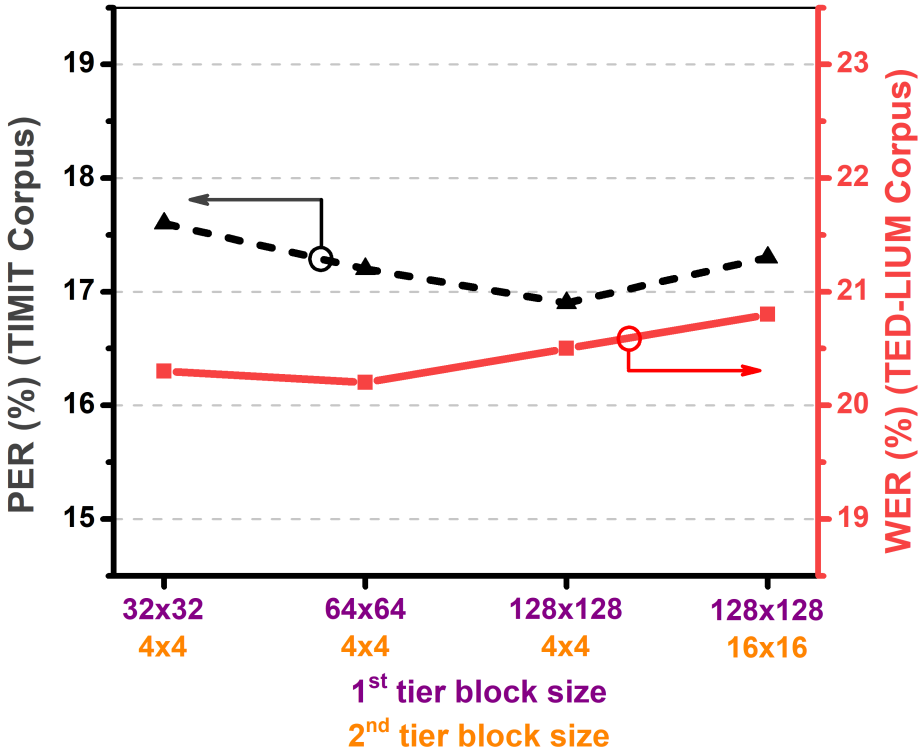



**Figure 5.5:** PER (TIMIT) and WER (TED-LIUM) values are shown for LSTMs trained with different HCGS block sizes. For all datapoints, compression rate is 16× and weight precision is 6-bit.

number of different LSTM configurations are evaluated, for the TIMIT corpus in Section **??** and the TED-LIUM corpus in Section 5.4.4.

### *5.4.2 HCGS Robustness to Block Size*

As mentioned in Section **??**, the chosen block size in the two tiers of HCGS affects the final accuracy of the network. In this work, we consider up to 1,024 LSTM cells, which means that the largest weight matrix size is 1,024×1,024. We constrain the smallest block size for the first tier to be 32×32, and the second tier block size to be within the range of 16×16 to 4×4.

With these constraints for the two tiers of HCGS, we evaluated all possible combinations of power-of-2 block sizes. Figure 5.5 shows PER (for TIMIT) and WER (for

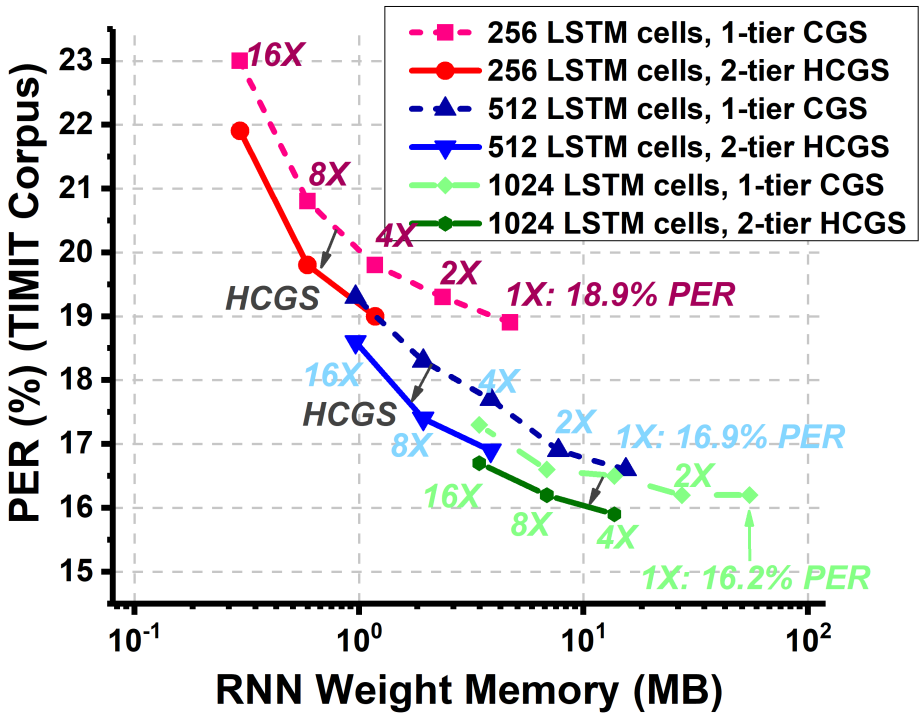



**Figure 5.6:** PER (TIMIT) comparison between single-tier CGS and two-tier HCGS schemes.

TED-LIUM) for different block sizes used for the two tiers of HCGS. It can be seen that LSTM training using HCGS with different block sizes results in similar PER and WER values, showing the robustness of HCGS-based LSTMs across a broad range of block size values.

### *5.4.3 Improvements due to HCGS*

We observe improvements in both accuracy and convergence time of the network when we train LSTMs with HCGS. Figure 5.6 shows the PER improvement due to the hierarchical structure in two-tier HCGS scheme, compared to the single-tier CGS scheme reproduced from Kadetotad *et al.* (2016). The results for LSTMs with different number of cells (256, 512, and 1,024) and compression rates (1× to 16×) are shown. In all experiments, LSTM networks trained with two-tier HCGS achieve lower PER than single-tier CGS for the same target compression.

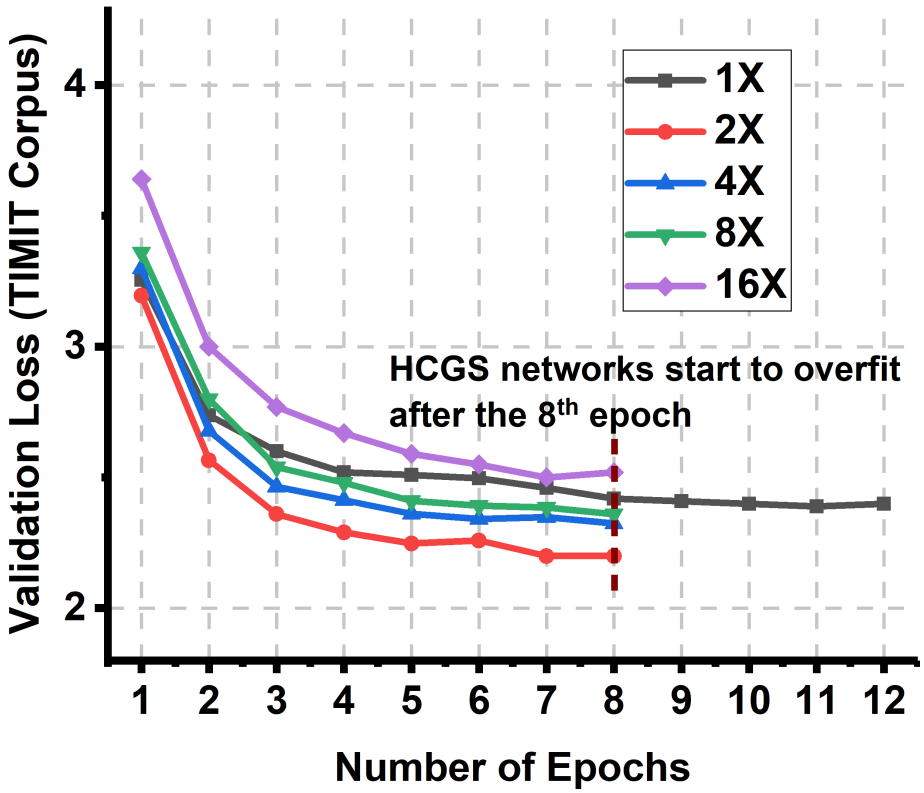



**Figure 5.7:** Network convergence comparison between various different target compression.

We believe the hierarchical sparsity leads to the improved accuracy of the networks. Sparse weights with fine granularity tend to form a uniform sparsity distribution even within smaller regions of the weight matrix. This property will lead to extremely sporadic and isolated connections when the target compression rate is high. However, the grouping of sparse weights within the hierarchical structure of HCGS allows densely connected regions to be formed even when the target compression rate is high. As two-tier HCGS-based LSTM networks outperform single-tier CGS in terms of accuracy, all datapoints reported in Section ?? and ?? are trained only with two-tier HCGS.

In addition, HCGS facilitates faster convergence of the network, thus requiring fewer epochs to train. This is shown in Figure 5.7, where the validation loss at the

end of each epoch is compared. We observe that sparse networks converge to the minima after 8 epochs while the uncompressed dense network requires 12 epochs to converge to the minima.

### *5.4.4 LSTM Results for TIMIT*

For TIMIT corpus, we trained 2-layer LSTMs for a number of different sizes (256, 512 and 1,024 LSTM cells), compression rates (2×, 4×, 8× and 16×) and weight quantization schemes (32-bit, 6-bit and 3-bit). Figure 5.8 shows the compiled PER and RNN weight memory curves of HCGS-based LSTMs for the TIMIT corpus. For a similar memory footprint, we observe that wider sparse networks perform better than narrower dense networks. This phenomenon is highlighted in Figure 5.8, where a 1,024-cell network with 8× compression shown a lower PER than a 512-cell network with 2× compression. Our observation conforms to observations reported in Wen *et al.* (2016); Zhu and Gupta (2017), where wider sparse networks performed better than narrow dense networks for similar weight memory requirements.

The Pareto frontier curve can be determined from Figure 5.8, which consists of data points that are at the most bottom or left part in the explored design space. LSTM networks corresponding to data points on the Pareto frontier curve offer the highest accuracy for the smallest memory footprint in the search space.

Figure 5.9 compares the total RNN weight memory requirement and PER reported for prior structured compression works Han *et al.* (2017); Wang *et al.* (2018); Li *et al.* (2018) for LSTM accelerators, with the Pareto optimal curve obtained with the proposed HCGS-based LSTMs. It can be seen that all points on the Pareto optimal curve of our proposed work provide lower PER while requiring less storage for the overall LSTM weights.

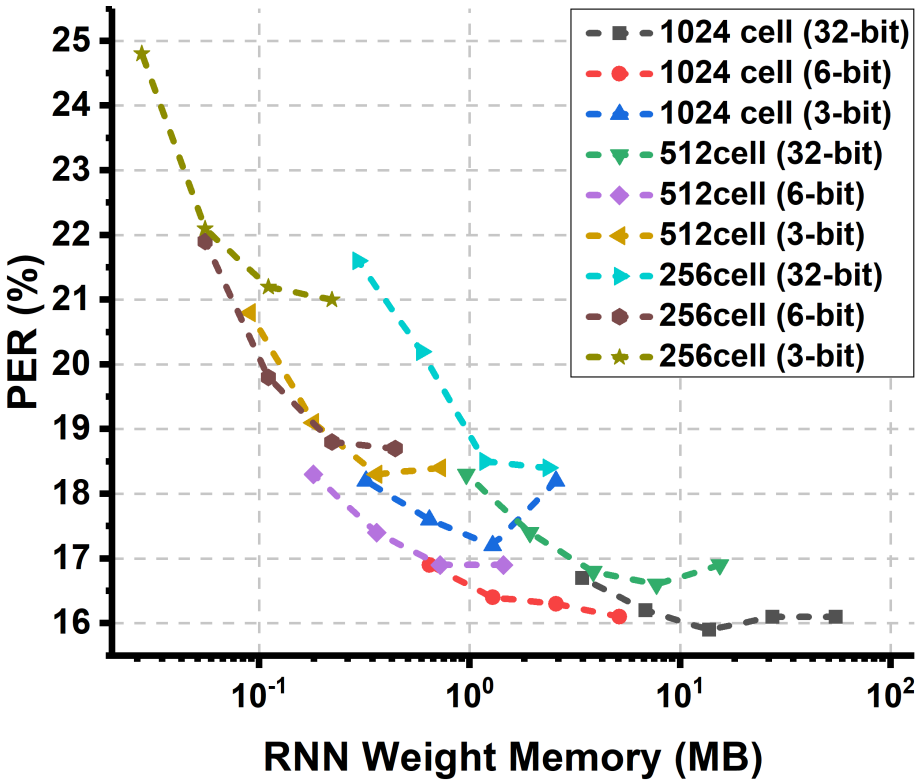



**Figure 5.8:** PER vs. RNN weight memory results for different sizes of 2-layer LSTMs for TIMIT, with various compression rates and quantization values.

### *5.4.5 LSTM Results for TED-LIUM*

For TED-LIUM corpus, we trained 3-layer LSTMs for a number of different sizes (256, 512 and 1,024 LSTM cells), compression rates (2×, 4×, 8×, 16×) and weight quantization schemes (32-bit, 6-bit, 3-bit). Figure 5.10 shows the compiled WER and RNN weight memory curves of HCGS-based LSTMs for the TED-LIUM corpus. We observe once more that for a similar memory footprint, wider sparse networks perform better than narrower dense networks. This phenomenon is highlighted again in Figure 5.10, where a 1,024-cell network with 8× compression results in a lower PER than a 512-cell network with 2× compression.

The Pareto frontier curve can be extracted from Figure 5.10 by choosing the data points that are at the most bottom and left parts in the search space. In Mishra

*et al.* (2018), it has been reported that wider CNNs can lower the precision of activations/weights much more than shallower counterparts, for the same or even better accuracy. However, in this work, we do not see such trends with LSTM RNNs for TIMIT or TED-LIUM. Especially when coupled together with structured compression, we find that LSTMs are more sensitive to low-precision quantization, so that LSTMs with medium (e.g. 6-bit) precision shows the best trade-off between PER/WER and memory, among the LSTMs we trained using in-training quantization.

Additional low-precision techniques for LSTM RNNs such as balanced quantization He *et al.* (2016) could be optimized together with the proposed HCGS, to investigate whether LSTMs could employ very low-precision quantization (e.g. 2-bit)

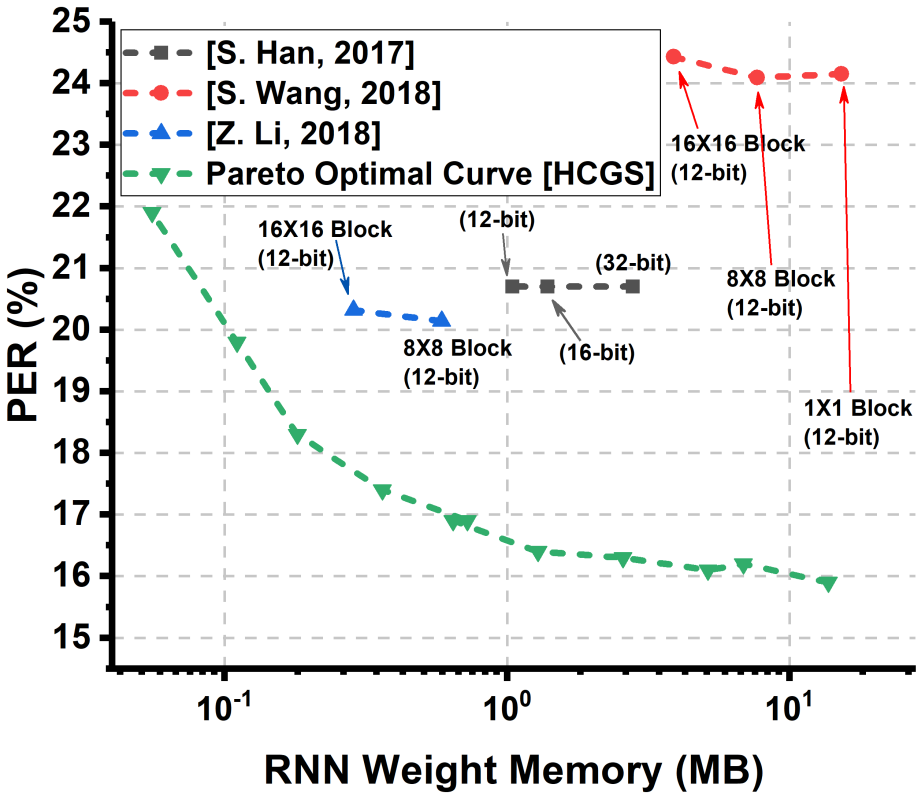



**Figure 5.9:** PER and weight memory are shown for prior LSTM compression works and the Pareto frontier curve obtained with HCGS-based LSTMs.

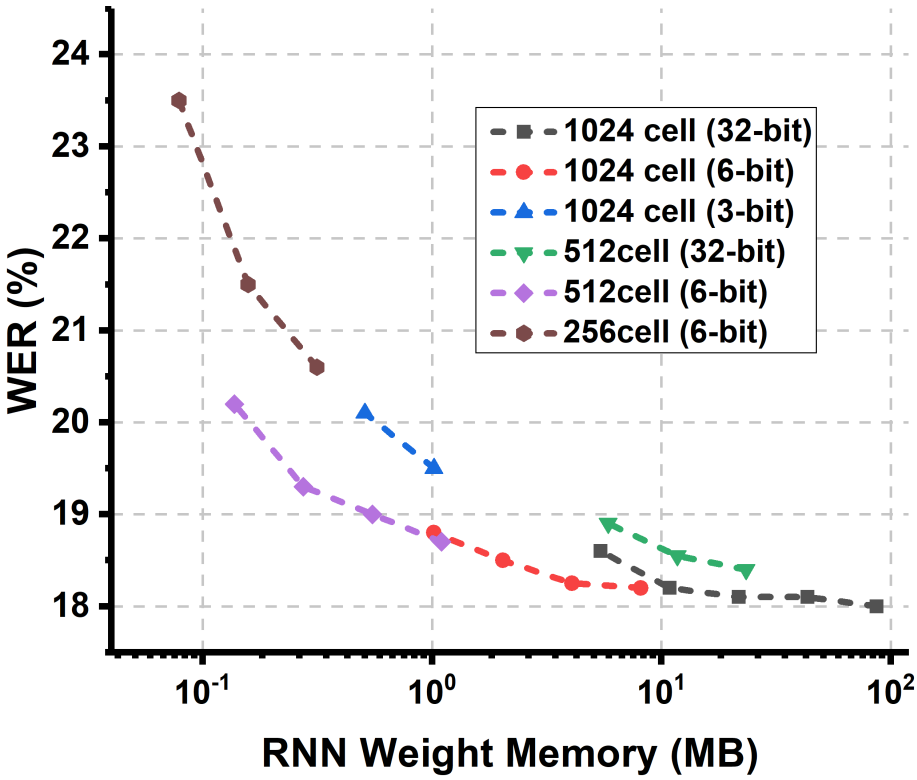



**Figure 5.10:** WER vs. RNN weight memory results for different sizes of 3-layer LSTMs for TED-LIUM, with various compression rates and quantization values.

without accuracy degradation, which remains as future work.

## 5.5 Discussion

Compressing LSTMs using column-/row-wise sparsity has been investigated in Wen *et al.* (2017). Column and row sparsity in LSTMs leads to removal of the neurons in the preceding and current layer, respectively. Removal of neurons will result in fewer weights to be stored but it inherently reduces the effective width of the LSTM network (e.g. number of LSTM cells), which can adversely affect the accuracy. Including our reported results, other researchers have also shown that wide sparse networks perform favorably than the narrow dense counterparts Wen *et al.* (2016); Zhu and Gupta (2017). Block-wise sparsity obtained through the proposed HCGS generates groups of randomly and sparsely connected neurons between layers, therefore not altering

the width of the LSTM network. Without reducing the number of LSTM cells, the proposed HCGS shows an effective way to substantially reduce the existing redundancy in the weight matrices and highly compress LSTMs, while minimizing accuracy degradation.

## 5.6 Conclusion

In this paper, we designed a new training algorithm for LSTMs, targeting hierarchical structured sparsity and low-precision quantization. Our training algorithm allows compression of the weights while reducing the index memory cost to a negligible value when compared to the compressed weights. Experiments conducted on both the TIMIT and TED-LIUM corpus demonstrated the effectiveness and general usability of HCGS across various LSTM RNNs. We also derived the Pareto optimal curve by jointly optimizing HCGS-based structured compression, low-precision quantization and the number of LSTM cells in RNNs. Our proposed HCGS methodology results in the best trade-off between accuracy and LSTM weight memory when compared to prior LSTM compression works.

**Algorithm 1** Training LSTM with HCGS. $\circ$ indicates element-wise multiplication, $C$ is the cost function for a minibatch, $\lambda$ is the learning rate decay factor, and $L$ is the number of layers.

**Require:** a minibatch of inputs and targets $(x, a^*)$, previous weights $W$ and $U$, HCGS mask $C^W$ and $C^U$ as well as previous learning rate $\eta$.

**Ensure:** updated weights $W^{t+1}$ and $U^{t+1}$ and updated learning rate $\eta^{t+1}$.

**Forward Propagation:**

**for** $k = 1$ to $L$ **do**

$W_{k_{i,f,o,c}} \leftarrow W_{k_{i,f,o,c}} \circ C_k^W$

$U_{k_{i,f,o,c}} \leftarrow U_{k_{i,f,o,c}} \circ C_k^U$

$h_{k,t} \leftarrow \text{Compute}(W_{k_{i,f,o,c}}, U_{k_{i,f,o,c}}, x_{k,t})$ {via (1)-(5)}

$x_{k+1,t} \leftarrow h_{k,t}$

**end for**

**Backward Propagation:**

$g_{W_{k_{i,f,o,c}}}$ and $g_{U_{k_{i,f,o,c}}}$ are the gradients calculated for each layer $k$ from 1 to $L$ and are represented below as $g_{W_k}$ and $g_{U_k}$ respectively for simplicity. Similarly $W_{k_{i,f,o,c}}$ and $U_{k_{i,f,o,c}}$ are represented as $W_k$ and $U_k$.

Parameter Update:

**for** $k = 1$ to $L$ **do**

$g_{W_k} \leftarrow g_{W_k} \circ C_k^W$

$W_k^{t+1} \leftarrow \text{Update}(W_k, \eta, g_{W_k})$

$g_{U_k} \leftarrow g_{U_k} \circ C_k^U$

$U_k^{t+1} \leftarrow \text{Update}(U_k, \eta, g_{U_k})$

$\eta^{t+1} \leftarrow \lambda\eta$

**end for**

**Algorithm 2** Quantization. $\circ$ indicates element-wise multiplication and / is element-wise division.

**Require:** weights $W$, quantize bits $n$.

$W \leftarrow clamp(W, -1, 1)$

$W^{sign} \leftarrow Sign(W)$

$W^{q_n} \leftarrow (\frac{ceil(abs(W) \circ 2^{n-1})}{2^{n-1}}) \circ W^{sign}$

Chapter 6

# A 8.93 TOPS/W LSTM RECURRENT NEURAL NETWORK ACCELERATOR FEATURING HIERARCHICAL COARSE-GRAIN SPARSITY WITH ALL PARAMETERS STORED ON-CHIP

Long short-term memory (LSTM) networks are widely used for speech applications but pose difficulties for efficient implementation on hardware due to large weight storage requirements. We present an energy-efficient LSTM recurrent neural network (RNN) accelerator, featuring an algorithm-hardware co-optimized memory compression technique called hierarchical coarse-grain sparsity (HCGS). Aided by HCGS-based block-wise recursive weight compression, we demonstrate LSTM networks with up to 16× fewer weights while achieving minimal accuracy loss. The prototype chip fabricated in 65nm LP CMOS achieves 8.93/7.22 TOPS/W for 2-/3-layer LSTM RNNs trained with HCGS for TIMIT/TED-LIUM corpora.

## 6.1 Introduction

The emergence of internet of things (IoT) devices that require edge computing with limited area and energy has garnered intense interest in energy-efficient ASIC accelerators for deep learning applications. The particular challenge of performing on-device automatic speech recognition (ASR) is that LSTMs that show high accuracy suffer from high complexity and require a large number of parameters to be trained and stored Xiong *et al.* (2018).

Recent works presented methods to reduce the complexity and storage of ASR hardware. Magnitude-based pruning was applied to LSTM hardware in Han *et al.* (2017), resulting in 20× model size reduction, but element-wise sparsity incurs con-

siderable index memory and irregular memory access, hurting both performance and power. To overcome this, structured sparsity techniques have been proposed with row-/column-wise sparsity for RNNs Wen *et al.* (2017), with block-wise sparsity for multi-layer perceptrons (MLPs) Kadetotad *et al.* (2016), and with block-circulant weight matrix for RNNs Wang *et al.* (2018) in speech processing applications. However, these works exhibit limited weight compression of ∼4× Wen *et al.* (2017); Kadetotad *et al.* (2016) or high error rate Wang *et al.* (2018), and have not been implemented in ASIC Han *et al.* (2017); Wen *et al.* (2017); Kadetotad *et al.* (2016); Wang *et al.* (2018). While recent ASIC designs targeting RNNs focus on improved energy-efficiency Conti *et al.* (2018); Yin *et al.* (2017b), they do not incorporate compression techniques and do not report RNN accuracy for representative benchmarks, which are both necessary to accomplish practical ASR on small-form-factor edge devices.

In this work, we present a new hierarchical coarse-grain sparsity (HCGS) scheme that structurely compresses LSTM weights by 16× with minimal accuracy loss. HCGS-based LSTM accelerator which executes 2-/3-layer LSTMs for real-time speech recognition was prototyped in 65nm LP CMOS. It consumes 1.85/3.42 mW power and achieves 8.93/7.22 TOPS/W for TIMIT/TED-LIUM corpora.

## 6.2 LSTM and Hierarchical Coarse-Grain Sparsity

### *6.2.1 LSTM-based Speech Recognition*

LSTM is a type of recurrent neural network (RNN) that shows state-of-the-art accuracy for speech recognition Xiong *et al.* (2018). Each layer of a LSTM consists of neurons, which computes the final output $h_t$ through four intermediate results called gates (Fig. 6.1). From the LSTM equations in Fig. 6.1, we see that the weight memory requirement of LSTMs is 8× when compared to MLPs with the same number

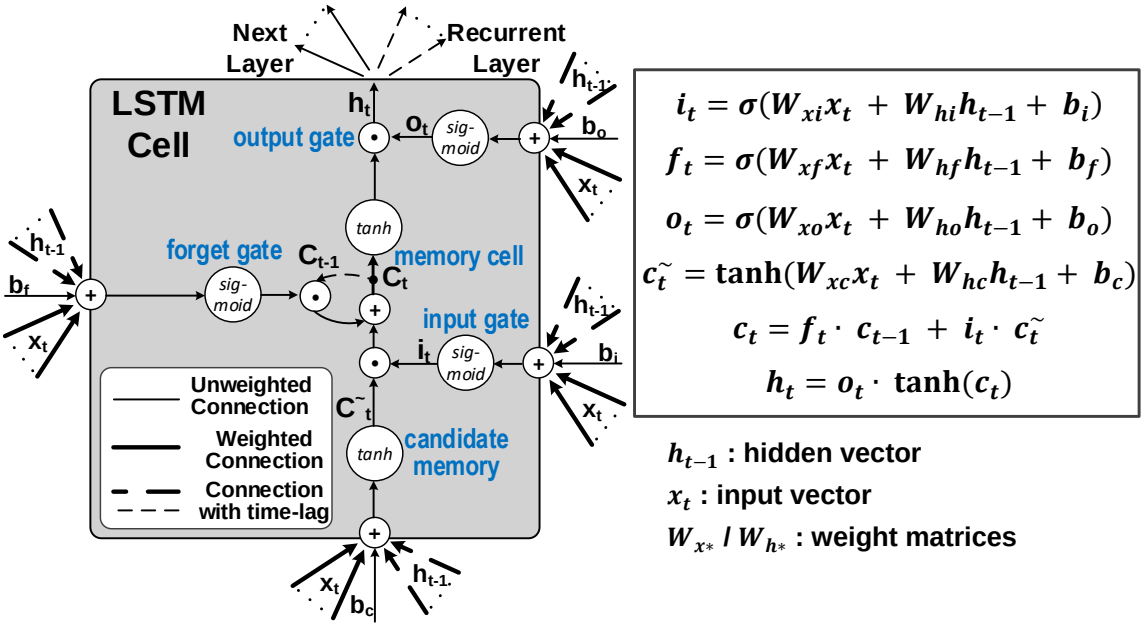



**Figure 6.1:** Illustration of LSTM cell with computation equations.

of neurons per layer.

LSTM-based speech recognition typically consists of a pipeline of a feature extraction module, followed by a LSTM RNN and then by a Viterbi decoder. A commonly used feature for speech recognition is feature-space Maximum Likelihood Linear Regression (fMLLR). fMLLR features are extracted from Mel Frequency Cepstral Coefficients (MFCC) features, derived typically from 25 ms windows of audio samples with a 10 ms overlap between subsequent windows. The features for the current window are combined with those of past and future windows to provide context and the merged set of features are inputs to the neural network. In our implementation, we merge five past and five future windows to the current window to create an input frame with 11 windows, leading to 440 fMLLR features per frame. The output layer of the LSTM consists of probability estimates that are sent to the Viterbi decoder unit to determine the best sequence of phonemes/words.

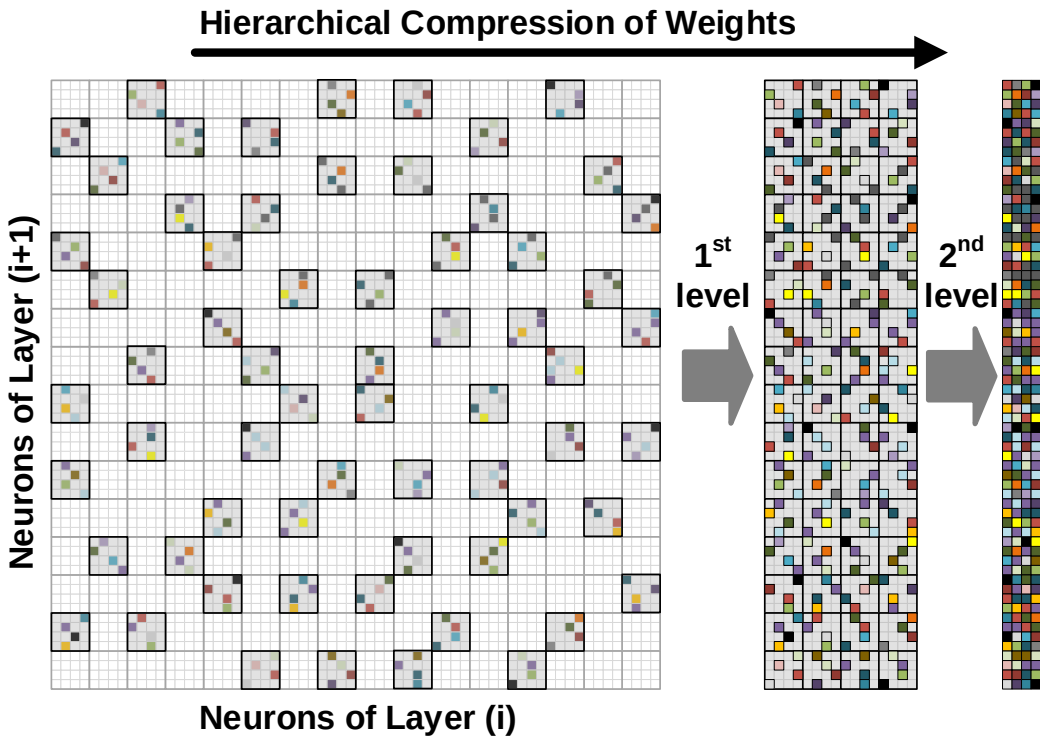



**Figure 6.2:** Illustration of LSTM RNN weight compression featuring the proposed hierarchical coarse-grain sparsity (HCGS).

### *6.2.2 Hierarchical Coarse-Grain Sparsity*

The proposed HCGS scheme maintains coarse-grain sparsity while further allowing fine-grain weight connectivity, leading to significant area and energy savings. Two-level HCGS is illustrated in Fig. 6.2, where the first level compresses weights (e.g. 4× compression) using a larger block size (e.g. 32×32) and the remaining weights in the large blocks go through the second level of compression (e.g. 4×) with a smaller block size (e.g. 8×8). The hierarchical structure of block-wise weights is randomly selected before the RNN training process starts, and is maintained throughout training and classification phases. The dropped blocks remain at zero and do not contribute to the physical memory footprint during both training and classification. TIMIT and TED-LIUM corpora are used to train the RNNs for phoneme and speech recognition, respectively. The baseline 3-layer, 512-cell LSTM RNN that performs speech

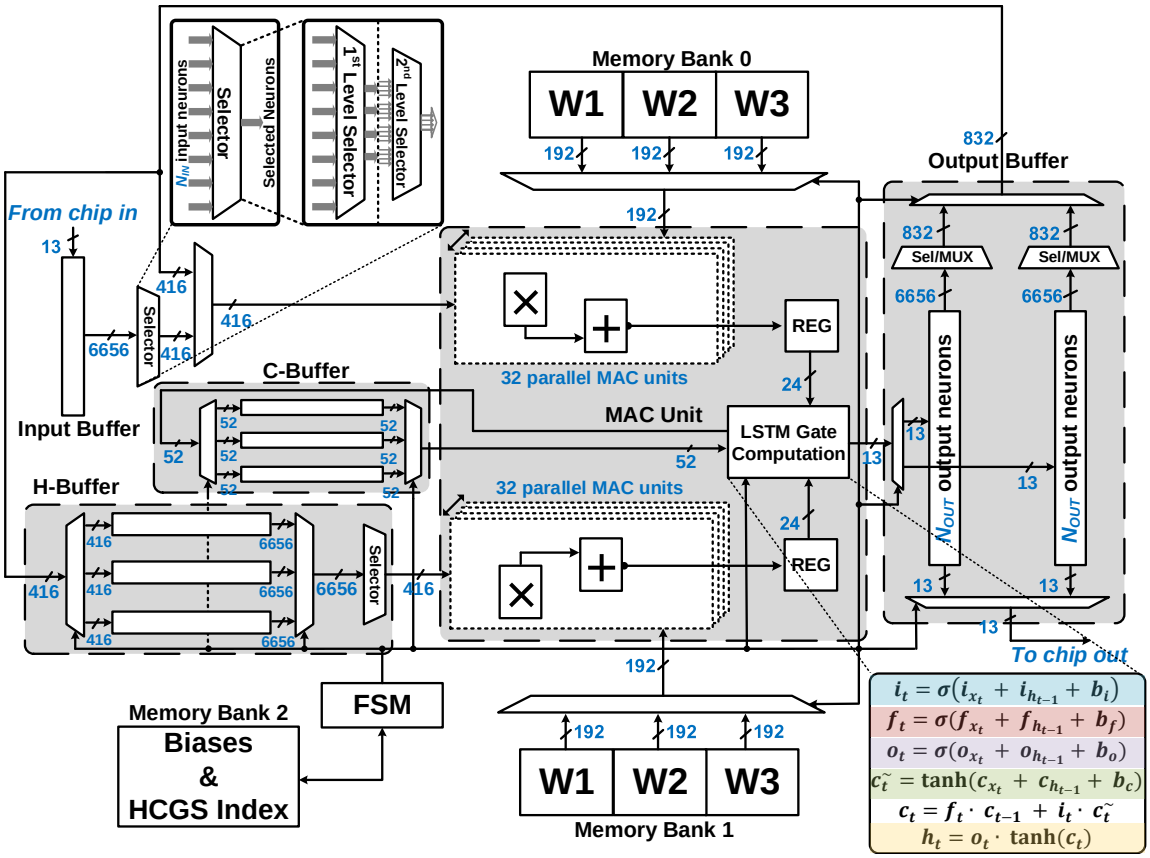



**Figure 6.3:** Overall architecture of the proposed LSTM RNN accelerator.

recognition for TED-LIUM corpus requires 24 MB of weight memory in floating-point precision. Aided by (1) the proposed HCGS that reduces the number of weights by 16× and (2) low-precision (6-bit) representation of weights, the compressed parameters of a 3-layer, 512-cell LSTM RNN are reduced to only 288 kB (83× reduction in model size compared to 24 MB). The resultant LSTM network can be fully stored on-chip, which results in energy-efficient acceleration.

## 6.3 Architecture and Design Optimizations

### *6.3.1 Hardware Architecture*

Fig. 6.3 shows the overall architecture of the proposed LSTM accelerator. It consists of the HCGS selector, input and output buffers, MAC unit, H-buffer, C-

buffer, two memory banks (144 kB each) for weight storage, bias/index memory bank (8.5 kB), and the global controller. The proposed architecture facilitates the computation of one LSTM cell output per cycle after an initial latency period and reuses the MAC unit as outputs are computed in a layer-by-layer manner.

### HCGS Selector

The HCGS selector (Fig. 6.3, top-left) has two levels, where the first level of selector only enables the propagation of activations associated with larger non-zero weights blocks and the second level further filters through the activations associated with smaller non-zero blocks. For 16× HCGS compression, only 32 activation outputs are required from a total of 512 activations, ensuring only activations corresponding to non-zero weights propagate to the MAC unit, greatly boosting energy-efficiency.

### Input and Output buffers

An input frame consists of fMLLR features as described in Sec. **??**. The input buffer is used to store the fMLLR features of an input frame, which is streamed in 13-bits at a time over 512 cycles. The output buffer consists of two identical buffers for double buffering, which enables continuous computation of the LSTM accelerator while streaming out the final layer outputs. Each output buffer employs a HCGS selector and a 6656:416 multiplexer to feedback the output of the current layer output to the next layer. The feedback from the output buffer to the input of the MAC facilitates the reuse of the MAC unit.

### H-buffer and C-buffer

The H-buffer and C-buffer store the outputs of the previous frame ($h_{t-1}$) and cell state ($c_{t-1}$) for each layer, respectively.

**MAC Unit**

The MAC unit consists of 64 parallel MACs (computing vector-matrix multiplications) and the LSTM gate computation module (computing intermediate LSTM gate and final output values), which can effectively perform 2,064 operations in each cycle aided by the proposed HCGS compression. The non-linear activation functions (sigmoid and hyperbolic tangent) are implemented through piece-wise linear modules using 20 linear segments.

**Weight/bias storage and global controller**

Weights are stored in the interleaved fashion as described in Sec. **??**, where each memory sub-bank (W1-W3) stores weights corresponding to a single layer. This allows sub-banks storing weights of layers not currently being computed to be in sleep mode, leading to improved energy-efficiency.

### *6.3.2 Interleaved Memory Storage*

Fig. 6.4 shows the LSTM module computation operations and detailed state diagram of the MAC unit in our LSTM accelerator. The LSTM cell stores the intermediate products to compute the cell state ($c_t$) and output ($h_t$). Conventionally the cell states and outputs of an entire layer are computed only after every intermediate gate output for the corresponding layer is completed, this leads to additional memory requirements to store the intermediate gate outputs for all the LSTM cells in the layer. Instead, by taking advantage of the structure of the LSTM cell, the proposed architecture cycles between the four states computing internal gates of the LSTM cell, namely input gate ($i_t$), forget gate ($f_t$), output gate ($o_t$), and candidate memory ($\tilde{c}_t$). Additionally, the vector-matrix multiplications of $x_t W_{x*}$ and $h_{t-1} W_{h*}$ can be computed in independent streams, which increases throughput via parallelism.

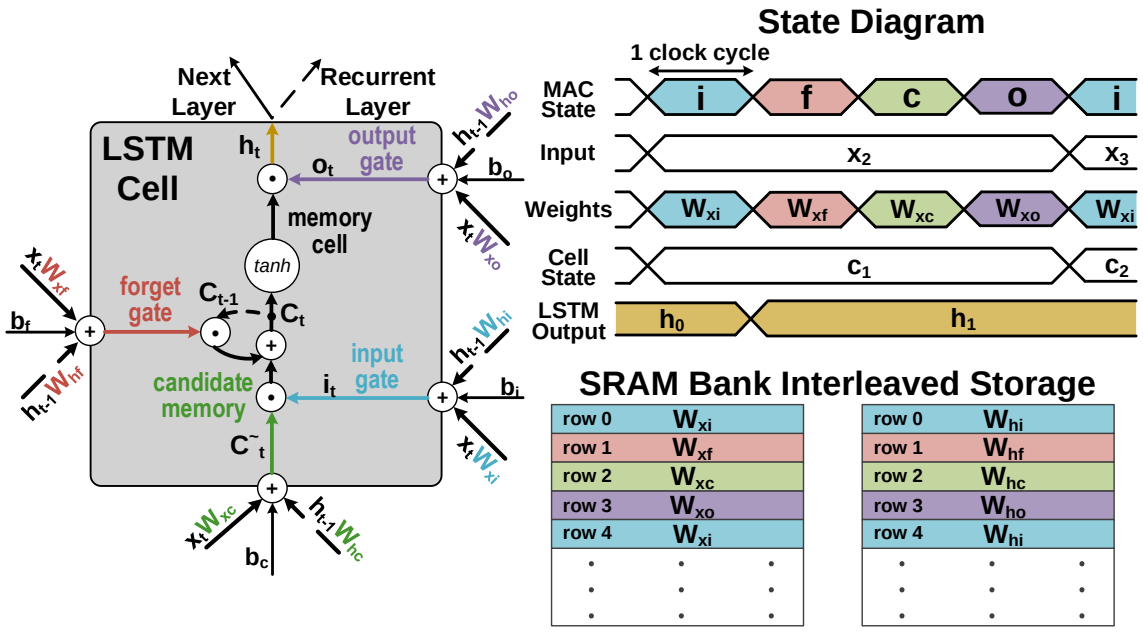



**Figure 6.4:** LSTM data flow and core computations.

To support this, we store each row of four matrices $W_{xi}$, $W_{xf}$, $W_{xo}$, and $W_{xc}$ in a staggered manner (same for $W_{h*}$) in on-chip SRAM (Fig. 6.4, right-bottom), so that the computation of new $c_t$ and $h_t$ values can be completed after every four cycles, hence eliminating the requirement to store all intermediate gate outputs of the layer. In addition, the same random hierarchical block selection for HCGS is applied to all four matrices of $W_{xi}$, $W_{xf}$, $W_{xo}$, and $W_{xc}$ (same for $W_{h*}$) to further reduce the index memory of the HCGS selector by 4×, resulting in only 1.17% index memory overhead.

### *6.3.3 Design Space Exploration*

There are several important design parameters for HCGS based LSTM hardware design, including HCGS block size, compression ratio, and random block assignments. For this design space exploration, we constructed a number of LSTM RNNs; the simulation results are summarized in Fig. 6.5. For our LSTM accelerator, we reduced the weight precision to 6-bit and activation precision to 13-bit with negligible accuracy loss. Compared to single-level CGS Kadetotad *et al.* (2016), the 2-level HCGS scheme

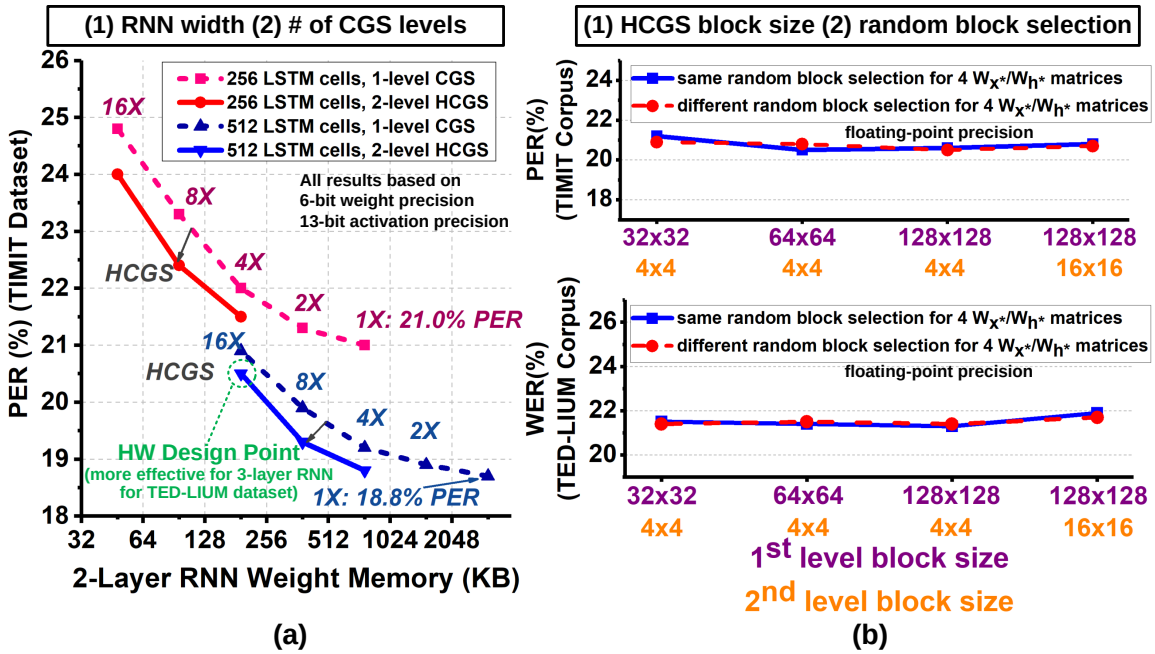



**Figure 6.5:** HCGS design space exploration. (a) RNN width and the number of CGS levels. (b) HCGS block size and random block selection.

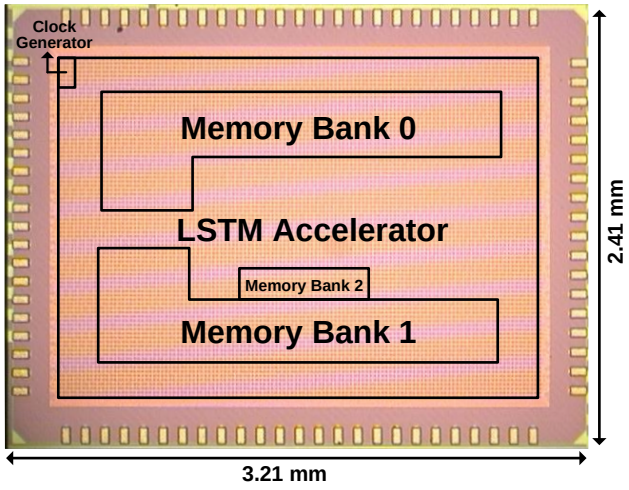
shows a favorable trade-off between phoneme error rate (PER) for TIMIT corpus and weight compression (Fig. 6.5(a)). Different random block assignments and sharing the HCGS masks for four LSTM gates do not affect the RNN accuracy (Fig. 6.5(b)). Overall, the 512-cell LSTMs shows better PER than the 256-cell LSTMs for various HCGS experiments. Based on these results, we selected the 512-cell LSTM and two-level HCGS to achieve up to 16× compression, for phoneme/speech recognition using TIMIT/TED-LIUM corpora.

## 6.4 Measurement & Comparison

The proposed LSTM RNN accelerator is fabricated in 65nm LP CMOS. The chip micrograph and performance summary are shown in Fig. 6.6. For chip testing, we initially load the weights, biases and configuration bits to on-chip memory. To verify real-time operation, 13-bit input fMLLR features are streamed into the input buffer, while RNN outputs from the chip are streamed out and stored.

Fig. 6.7 shows the chip measurement results, where the accelerator operates up to 80MHz at 1.1V while consuming 67.3/72.5 mW for 2-/3-layer LSTMs, respectively. With voltage scaling, the power consumption at 0.68V for the 2-layer RNN for TIMIT is 1.85 mW at 8 MHz (Fig. 6.7(a)), and at 0.75V for the 3-layer RNN for TED-LIUM is 3.42 mW at 12 MHz (Fig. 6.7(b)). In all cases, the accelerator satisfies the real-time speech/phoneme recognition requirement of 100 frames/second. The memory and logic power breakdown for the 3-layer RNN at 0.75V is shown in Fig. 6.7(d). It can be seen that logic power is dominant due to the highly compressed weight memory despite the large number of RNN weight matrices. Pipelined with the LSTM gate computation unit, the MAC engines exhibit a high utilization ratio of 99.66%.

By leveraging HCGS, the LSTM accelerator achieves average energy-efficiency of 8.93/7.22 TOPS/W for running end-to-end 2-/3-layer LSTM RNNs for TIMIT/TED-LIUM corpora (Fig. 6.7(c)). We report the measured accuracy results of 20.6% PER for TIMIT and 21.3% word error rate (WER) for TED-LIUM in Fig. 6.8. Compared to the RNN ASIC works of Conti *et al.* (2018) and Yin *et al.* (2017b), this work shows 2.90× and 1.75× higher energy-efficiency (TOPS/W), respectively. Table 6.1 shows




| Technology | 65nm LP CMOS | |
|---|---|---|
| On-chip SRAM | 288 KB (weights)<br>8.5 KB (index & bias) | |
| Supply Voltage | 0.68V – 1.1V | |
| RNN Precision | 6-bit (weights)<br>13-bit (activations) | |
| Speech Dataset | TIMIT | TED-LIUM |
| # of RNN Layers | 2 | 3 |
| Error Rate | PER=20.6% | WER=21.3% |
| Power (@ real-time perf.)<br>Energy-Efficiency<br>(Vdd, Freq.) | 1.85 mW<br>8.93 TOPS/W<br>(0.68V, 8MHz) | 3.42 mW<br>7.24 TOPS/W<br>(0.75V, 12MHz) |

**Figure 6.6:** Prototype chip micrograph and performance summary.

the detailed comparison with prior ASIC/FPGA works for RNNs.

Fig. 6.8 shows a comparison of frames/second/power (FPS/W) and PER for TIMIT corpus with prior works Han *et al.* (2017); Wang *et al.* (2018); Esser *et al.* (2016) that perform speech/phoneme recognition. The RNN accelerator **?** reports low power consumption but can only support limited keyword spotting tasks and is not considered. Compared to 28nm ASIC design supporting speech recognition Esser *et al.* (2016), this work shows 2.95× higher energy-efficiency (FPS/W) with slightly better PER. Although FPS/W in Wang *et al.* (2018) is comparable to our work, we achieve considerably lower PER. Conversely, Han *et al.* (2017) has comparable PER to our work but poor FPS/W. Overall, this demonstrates the effectiveness of our proposed design due to the algorithm-hardware co-optimization.

## 6.5 Conclusion

This paper presents a hierarchically compressed, energy-efficient LSTM accelerator for speech recognition. Exploiting the hierarchical block-wise sparsity and low-precision quantization, the accelerator stores the entire compressed weights of 3-layer, 512-cell LSTMs in 288 kB of on-chip SRAM and reduces the required computation by up to 16×. The 65nm prototype chip achieves average energy-efficiency of 8.93/7.22 TOPS/W for 2-/3-layer LSTMs for TIMIT/TED-LIUM corpora.

**Table 6.1:** Comparison of RNN performance with prior works.

| | **Han *et al.* (2017)** | **Wang *et al.* (2018)** | **Conti *et al.* (2018)** | **Yin *et al.* (2017b)** | **This Work** |
|---|---|---|---|---|---|
| **Technology** | FPGA | FPGA | 65nm CMOS | 65nm CMOS | 65nm CMOS |
| **Area ($mm^2$)** | - | - | 1.57 | 19.36 | 7.74 |
| **On-Chip Memory (KB)** | 4.2 MB | 280 | 82 | 348 | 297 |
| **Number of MACs** | - | - | 96 | - | 65 |
| **Bit-Precision Weights / Activations** | 12/16 | 16/16 | 8/16 | 16/16 | 6/13 |
| **Core Voltage (V)** | - | - | 1.24/0.75 | 1.2/0.67 | 1.1/0.68 |
| **Frequency (MHz)** | 200 | 200 | 168/20 | 200/10 | 80/8 |
| **Power (mW)** | 41W | 22W | 29/1.2 | 447/4 | 67.3/1.85 |
| **Peak Performance (GOPS)** | 2500 | - | - | - | 164.95/24.60 |
| **Energy-Efficiency (TOPS/W)** | 0.061 | 2.08 | 1.11/3.08 | 1.06/5.09 | 2.45/8.93 |
| **PER (TIMIT)** | 20.7% | 25.3% | - | - | 20.6% (measured) |
| **WER (TED-LIUM)** | - | - | - | - | 21.3% (measured) |

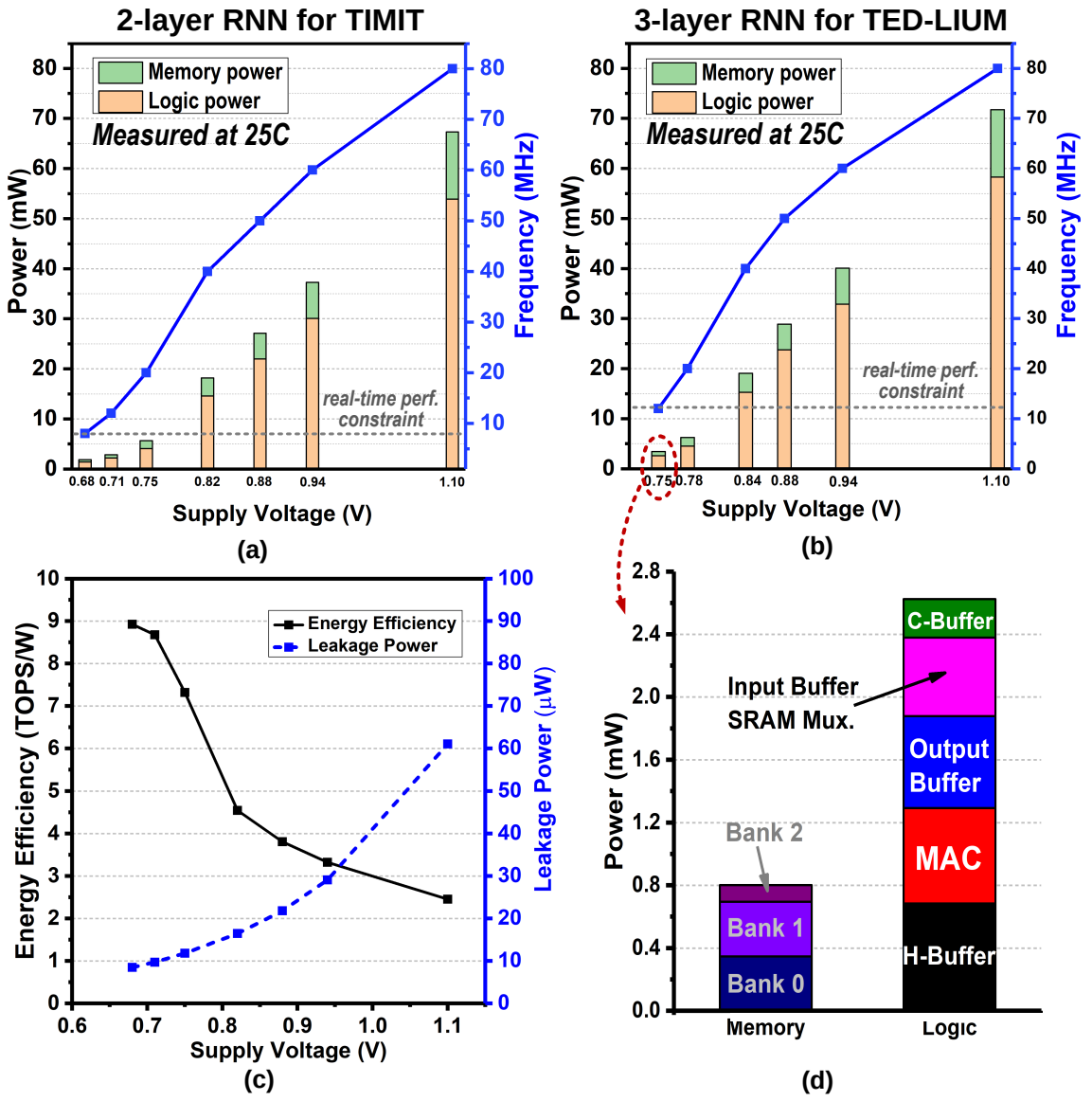
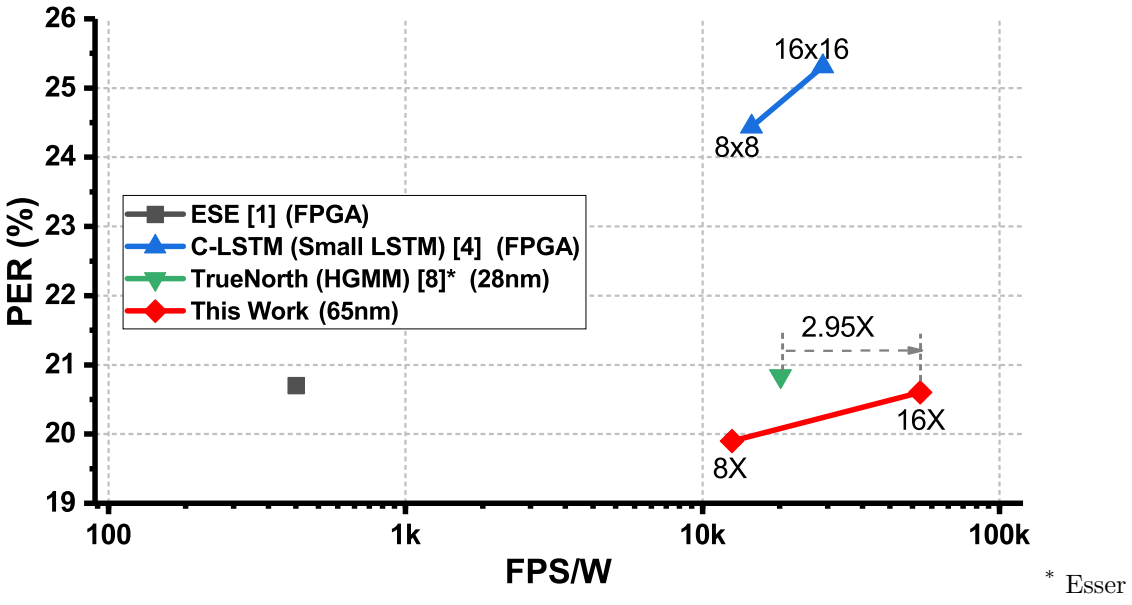



**Figure 6.7:** Power and frequency measurement results with voltage scaling for (a) 2-layer LSTM for TIMIT and (b) 3-layer LSTM for TED-LIUM. (c) Measurement results for energy-efficiency (TOPS/W) and leakage power. (d) Power breakdown of 3-layer LSTM at 0.75V supply.




* Esser *et al.* (2016) classification error from HGMM.

**Figure 6.8:** Comparison of TIMIT PER and energy efficiency (frames per second/power, FPS/W) with prior LSTM implementations.

# Chapter 7

# CONCLUSION

Power efficient implementation of deep neural network (DNN) hardware has become crucial for mobile devices. The surge in applications that use image based recognition, reconstruction or segmentation operations as well as speech based keyword or text conversion has forced alteration of both the algorithm and hardware implementation methods use to compute DNN outputs. As neural networks hardware implementation is both very compute and memory intensive, introducing sparsity will alleviate both challenges simultaneously.

The central operation for any neural network is the summing of products of weighted inputs. This process when looked at through the lense of conventional von Neumann architectures, require the reading the weights from a memory structure (e.g. SRAM, DRAM) and multiply them with corresponding inputs, which has been shown to be inefficient. Towards this need, our first contribution is an accelerator for matrix vector multiplication (Kadetotad *et al.* (2014, 2015, 2017); Chen *et al.* (2015); Xu *et al.* (2014)). Using resistive non-volatile memory to store and encode the weights with the conductance of the memory cells, the designed peripheral circuits enable power efficient implementation of sum-of-products. The proposed method has been shown to speedup the operation of sparse coding by up to 3000×.

Following the acceleration of matrix multiplication in sparse coding we focused on relieving the high cost of storing weights through block-wise sparsity in the scheme called coarse grain sparsity (CGS) (Kadetotad *et al.* (2016); Yin *et al.* (2017a); Srivastava *et al.* (2019)). The CGS framework has been demonstrated to be hardware friendly due to the block-wise sparsity and provides definite control of sparsity to the

user. Further more the CGS scheme for hardware implementation in monolithic 3D integrated circuits was explored in Chang *et al.* (2016b, 2018), where experiments conducted show power savings when using CGS trained DNNs.

CGS trained DNNs have shown to reduce weight storage requirements as well as total number of computations as the operations pertaining to the zero weights are now skipped. Therefore further increasing the sparsity through hierarchical coarse grain sparsification (HCGS) will allow for smaller granularity, leading to greater power savings. The algorithm implementation of HCGS is shown in Chapter 5 where different sparsity levels are swept to find a hardware design point for the 65nm prototype chip in Chapter 6. Applying HCGS provides greater sparsity than CGS while also showing better accuracy. Therefore, using the algorithm-hardware co-design we achieve state-of-the-art energy efficiency of 8.93TOPS/W for the 65nm chip.

# REFERENCES


Abbott, L. F., "Lapicques introduction of the integrate-and-fire model neuron (1907)", Brain research bulletin **50**, 5-6, 303–304 (1999).

Afifi, A., A. Ayatollahi, F. Raissi and H. Hajghassem, "Efficient hybrid cmos-nano circuit design for spiking neurons and memristive synapses with stdp", IEICE transactions on fundamentals of electronics, communications and computer sciences **93**, 9, 1670–1677 (2010).

Amodei, D., S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin", in "Proceedings of the 33rd International Conference on Machine Learning (ICML)", pp. 173–182 (2016).

Batude, P., M. Vinet, A. Pouydebasque, C. L. Royer, B. Previtali, C. Tabone, J. M. Hartmann, L. Sanchez, L. Baud, V. Carron, A. Toffoli, F. Allain, V. Mazzocchi, D. Lafond, O. Thomas, O. Cueto, N. Bouzaida, D. Fleury, A. Amara, S. Deleonibus and O. Faynot, "Advances in 3D CMOS Sequential Integration", (2009).

Bi, G.-q. and M.-m. Poo, "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type", Journal of neuroscience **18**, 24, 10464–10472 (1998).

Bottou, L. and O. Bousquet, "The tradeoffs of large scale learning", in "Advances in neural information processing systems", pp. 161–168 (2008).

Bradley, A. P., "The use of the area under the roc curve in the evaluation of machine learning algorithms", Pattern recognition **30**, 7, 1145–1159 (1997).

Canning, A. and E. Gardner, "Partially connected models of neural networks", Journal of Physics A: Mathematical and General **21**, 15, 3275 (1988).

Chang, K., D. Kadetotad, Y. Cao, J. Seo and S. K. Lim, "Monolithic 3D IC Designs for Low-Power Deep Neural Networks Targeting Speech Recognition", (2017).

Chang, K., D. Kadetotad, Y. Cao, J.-S. Seo and S. K. Lim, "Power, performance, and area benefit of monolithic 3d ics for on-chip deep neural networks targeting speech recognition", ACM Journal on Emerging Technologies in Computing Systems (JETC) **14**, 4, 42 (2018).

Chang, K., S. Sinha, B. Cline, R. Southerland, M. Doherty, G. Yeric and S. K. Lim, "Cascade2D: A Design-aware Partitioning Approach to Monolithic 3D IC with 2D Commercial Tools", (2016a).

Chang, K., S. Sinha, B. Cline, G. Yeric and S. K. Lim, "Match-making for Monolithic 3D IC: Finding the Right Technology Node", (2016b).

Chen, P.-Y., D. Kadetotad, Z. Xu, A. Mohanty, B. Lin, J. Ye, S. Vrudhula, J.-s. Seo, Y. Cao and S. Yu, "Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip", in "Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition", pp. 854–859 (EDA Consortium, 2015).

Chen, Y.-H., T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks", IEEE Journal of Solid-State Circuits **52**, 1, 127–138 (2017).

Cheng, Y., D. Wang, P. Zhou and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks", (2017).

Cheng, Y., F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary and S.-F. Chang, "An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections", in "Proc. IEEE Int. Conf. on Computer Vision", (2015).

Coates, A. and A. Y. Ng, "The importance of encoding versus training with sparse coding and vector quantization", in "Proceedings of the 28th international conference on machine learning (ICML-11)", pp. 921–928 (2011).

Conneau, A., D. Kiela, H. Schwenk, L. Barrault and A. Bordes, "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data", (2017).

Conti, F., L. Cavigelli, G. Paulin, I. Susmelj and L. Benini, "Chipmunk: A systolically scalable 0.9 mm 2, 3.08 gop/s/mw@ 1.2 mw accelerator for near-sensor recurrent neural network inference", in "Custom Integrated Circuits Conference (CICC), 2018 IEEE", pp. 1–4 (IEEE, 2018).

Courbariaux, M., Y. Bengio and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations", in "Advances in neural information processing systems", pp. 3123–3131 (2015).

Courbariaux, M., I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to+ 1 or-1", (2016).

Daubechies, I., M. Defrise and C. De Mol, "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint", Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences **57**, 11, 1413–1457 (2004).

Deng, L., G. Hinton and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview", in "Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on", pp. 8599–8603 (IEEE, 2013a).

Deng, L., J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. L. Seltzer, G. Zweig, X. He, J. D. Williams *et al.*, "Recent advances in deep learning for speech research at microsoft.", in "ICASSP", vol. 26, p. 64 (2013b).

Esser, S. K. *et al.*, "From the cover: Convolutional networks for fast, energy-efficient neuromorphic computing", Proceedings of the National Academy of Sciences of the United States of America **113**, 41, 11441 (2016).

Gales, M. J. *et al.*, "Maximum likelihood linear transformations for HMM-based speech recognition", Computer Speech & Language **12**, 2, 75–98 (1998).

Gardner, W. A., "Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique", Signal processing **6**, 2, 113–133 (1984).

Garofolo, J., L. Lamel, W. Fisher, J. Fiscus, D. Pallett and N. Dahlgren, "The DARPA TIMIT acoustic-phonetic continuous speech corpus", National Institute of Standards and Technology (1990).

Garofolo, J. S., L. F. Lamel, W. M. Fisher, J. G. Fiscus and D. S. Pallett, "DARPA TIMIT Acoustic-Phonetic Continous Speech Corpus", NASA STI/Recon Technical Report N (1993).

Gokhale, V., J. Jin, A. Dundar, B. Martini and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks", in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops", pp. 682–687 (2014).

Graves, A., A.-r. Mohamed and G. Hinton, "Speech recognition with deep recurrent neural networks", in "Acoustics, speech and signal processing (icassp), 2013 ieee international conference on", pp. 6645–6649 (IEEE, 2013).

Gray, S., A. Radford and D. Kingma, "Gpu Kernels for Block-Sparse Weights", Tech. rep., OpenAI (2017).

Gupta, S., A. Agrawal, K. Gopalakrishnan and P. Narayanan, "Deep learning with limited numerical precision", in "International Conference on Machine Learning", pp. 1737–1746 (2015).

Halpern, M., Y. Zhu and V. J. Reddi, "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction", in "Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)", pp. 64–76 (2016).

Han, K. J., A. Chandrashekaran, J. Kim and I. Lane, "The CAPIO 2017 conversational speech recognition system", arXiv preprint arXiv:1801.00059 (2018).

Han, S., J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA", in "Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays", (2017).

Han, S., H. Mao and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding", arXiv preprint arXiv:1510.00149 (2015a).

Han, S., H. Mao and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding", in "Proc. Int. Conf. on Learning Representations", (2016).

Han, S., J. Pool, J. Tran and W. Dally, "Learning both weights and connections for efficient neural network", in "Advances in Neural Information Processing Systems", pp. 1135–1143 (2015b).

He, K., X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition", in "Proc. IEEE Conf. on Computer Vision and Pattern Recognition", (2016).

He, Q., H. Wen, S. Zhou, Y. Wu, C. Yao, X. Zhou and Y. Zou, "Effective quantization methods for recurrent neural networks", arXiv preprint arXiv:1611.10176 (2016).

He, T., Y. Fan, Y. Qian, T. Tan and K. Yu, "Reshaping deep neural network for fast decoding by node-pruning", in "Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on", pp. 245–249 (IEEE, 2014).

Hochreiter, S. and J. Schmidhuber, "Long short-term memory", Neural Computation **9**, 8, 1735–1780 (1997).

Hoyer, P. O., "Non-negative sparse coding", in "Neural Networks for Signal Processing, 2002. Proceedings of the 2002 12th IEEE Workshop on", pp. 557–565 (IEEE, 2002).

Hubara, I., M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations.", Journal of Machine Learning Research **18**, 187, 1–30 (2017).

Ioffe, S. and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in "Proceedings of the 32nd International Conference on Machine Learning (ICML)", pp. 448–456 (2015).

Jo, S. H., T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems", Nano letters **10**, 4, 1297–1301 (2010).

Kadetotad, D., S. Arunachalam, C. Chakrabarti and J. Seo, "Efficient Memory Compression in Deep Neural Networks using Coarse-Grain Sparsification for Speech Applications", (2016).

Kadetotad, D., P.-Y. Chen, Y. Cao, S. Yu and J.-s. Seo, "Peripheral circuit design considerations of neuro-inspired architectures", in "Neuro-inspired Computing Using Resistive Synaptic Devices", pp. 167–182 (Springer, 2017).

Kadetotad, D., Z. Xu, A. Mohanty, P.-Y. Chen, B. Lin, J. Ye, S. Vrudhula, S. Yu, Y. Cao and J.-s. Seo, "Neurophysics-inspired parallel architecture with resistive crosspoint array for dictionary learning", in "Biomedical Circuits and Systems Conference (BioCAS), 2014 IEEE", pp. 536–539 (IEEE, 2014).

Kadetotad, D., Z. Xu, A. Mohanty, P.-Y. Chen, B. Lin, J. Ye, S. Vrudhula, S. Yu, Y. Cao and J.-s. Seo, "Parallel architecture with resistive crosspoint array for dictionary learning acceleration", IEEE Journal on Emerging and Selected Topics in Circuits and Systems **5**, 2, 194–204 (2015).

Karpathy, A. and L. Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", in "Proc. IEEE Conf. on Computer Vision and Pattern Recognition", (2015).

Katyal, V., R. L. Geiger and D. J. Chen, "Adjustable hysteresis cmos schmitt triggers", in "Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on", pp. 1938–1941 (IEEE, 2008).

Kim, J. K., P. Knag, T. Chen and Z. Zhang, "A 6.67 mw sparse coding asic enabling on-chip learning and inference", in "VLSI Circuits Digest of Technical Papers, 2014 Symposium on", pp. 1–2 (IEEE, 2014a).

Kim, J. K., P. Knag, T. Chen and Z. Zhang, "Efficient hardware architecture for sparse coding", IEEE Transactions on Signal Processing **62**, 16, 4173–4186 (2014b).

Krizhevsky, A., I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in "Advances in neural information processing systems", pp. 1097–1105 (2012).

Kuzum, D., S. Yu and H. P. Wong, "Synaptic electronics: materials, devices and applications", Nanotechnology **24**, 38, 382001 (2013).

Laurent, C., G. Pereyra, P. Brakel, Y. Zhang and Y. Bengio, "Batch normalized recurrent neural networks", in "Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)", pp. 2657–2661 (2016).

LeCun, Y., C. Cortes and C. Burges, "Mnist handwritten digit database", AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist **2** (2010).

Lee, J., J. Lee, D. Han, J. Lee, G. Park and H.-J. Yoo, "7.7 lnpu: A 25.3 tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16", in "2019 IEEE International Solid-State Circuits Conference-(ISSCC)", pp. 142–144 (IEEE, 2019).

Lee, J., J. Shin, D. Lee, W. Lee, S. Jung, M. Jo, J. Park, K. P. Biju, S. Kim, S. Park *et al.*, "Diode-less nano-scale zro x/hfo x rram device with excellent switching uniformity and reliability for high-density cross-point memory applications", in "Electron Devices Meeting (IEDM), 2010 IEEE International", pp. 19–5 (IEEE, 2010).

Li, Z., S. Wang, C. Ding, Q. Qiu, Y. Wang and Y. Liang, "Efficient recurrent neural networks using structured matrices in FPGAs", arXiv preprint arXiv:1803.07661 (2018).

Liang, J., S. Yeh, S. S. Wong and H.-S. P. Wong, "Effect of wordline/bitline scaling on the performance, energy consumption, and reliability of cross-point memory array", ACM Journal on Emerging Technologies in Computing Systems (JETC) **9**, 1, 9 (2013).

Liao, S., Z. Li, X. Lin, Q. Qiu, Y. Wang and B. Yuan, "Energy-Efficient, High-Performance, Highly-Compressed Deep Neural Network Design using Block-Circulant Matrices", (2017).

Liao, W., L. He and K. M. Lepak, "Temperature and Supply Voltage aware Performance and Power Modeling at Microarchitecture Level", **24**, 7, 1042–1053 (2005).

Lin, B., Q. Li, Q. Sun, M.-J. Lai, I. Davidson, W. Fan and J. Ye, "Stochastic coordinate coding and its application for drosophila gene expression pattern annotation", arXiv preprint arXiv:1407.8147 (2014).

Louizos, C., K. Ullrich and M. Welling, "Bayesian compression for deep learning", in "Advances in Neural Information Processing Systems", pp. 3288–3298 (2017).

Mairal, J., F. Bach, J. Ponce and G. Sapiro, "Online dictionary learning for sparse coding", in "Proceedings of the 26th annual international conference on machine learning", pp. 689–696 (ACM, 2009).

Mead, C. and M. Ismail, *Analog VLSI implementation of neural systems*, vol. 80 (Springer Science & Business Media, 2012).

Miao, Y., "Kaldi+ pdnn: building dnn-based asr systems with kaldi and pdnn", arXiv preprint arXiv:1401.6984 (2014).

Mishra, A., E. Nurvitadhi, J. J. Cook and D. Marr, "WRPN: Wide reduced-precision networks", in "Proceedings of the International Conference on Learning Representations (ICLR)", (2018).

Narang, S., E. Elsen, G. Diamos and S. Sengupta, "Exploring sparsity in recurrent neural networks", arXiv preprint arXiv:1704.05119 (2017).

Nayak, D. K., S. Banna, S. K. Samal and S. K. Lim, "Power, Performance, and Cost Comparisons of Monolithic 3D ICs and TSV-based 3D ICs", in "IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conf.", (2015).

Olshausen, B. A. and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images", Nature **381**, 6583, 607 (1996).

Panth, S., K. Samadi, Y. Du and S. K. Lim, "Design and CAD Methodologies for Low Power Gate-level Monolithic 3D ICs", (2014).

Park, S., A. Sheri, J. Kim, J. Noh, J. Jang, M. Jeon, B. Lee, B. Lee, B. Lee and H. Hwang, "Neuromorphic speech systems using advanced reram-based synapse", in "Electron Devices Meeting (IEDM), 2013 IEEE International", pp. 25–6 (IEEE, 2013).

Povey, D., A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz *et al.*, "The kaldi speech recognition toolkit", in "IEEE 2011 workshop on automatic speech recognition and understanding", No. EPFL-CONF-192584 (IEEE Signal Processing Society, 2011).

Price, P., W. M. Fisher, J. Bernstein and D. S. Pallett, "The darpa 1000-word resource management database for continuous speech recognition", in "Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on", pp. 651–654 (IEEE, 1988).

Rajendran, B., Y. Liu, J.-s. Seo, K. Gopalakrishnan, L. Chang, D. J. Friedman and M. B. Ritter, "Specifications of nanoscale devices and circuits for neuromorphic computational systems", IEEE Transactions on Electron Devices **60**, 1, 246–253 (2013).

Rath, S. P., D. Povey, K. Veselỳ and J. Cernockỳ, "Improved feature processing for deep neural networks.", in "Interspeech", pp. 109–113 (2013).

Ravanelli, M., P. Brakel, M. Omologo and Y. Bengio, "Improving speech recognition by revising gated recurrent units", arXiv preprint arXiv:1710.00641 (2017).

Ravanelli, M., T. Parcollet and Y. Bengio, "The PyTorch-Kaldi speech recognition toolkit", arXiv preprint arXiv:1811.07453 (2018).

Robinson, T., M. Hochberg and S. Renals, "The use of recurrent neural networks in continuous speech recognition", in "Automatic speech and speaker recognition", pp. 233–258 (Springer, 1996).

Rohlicek, J. R., W. Russell, S. Roukos and H. Gish, "Continuous hidden markov modeling for speaker-independent word spotting", in "Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on", pp. 627–630 (IEEE, 1989).

Rosenblatt, F., "Principles of neurodynamics. perceptrons and the theory of brain mechanisms", Tech. rep., Cornell Aeronautical Lab Inc Buffalo NY (1961).

Rousseau, A., P. Deleglise and Y. Esteve, "TED-LIUM: an automatic speech recognition dedicated corpus", in "Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)", (2012).

Rumelhart, D. E., G. E. Hinton and R. J. Williams, "Learning internal representations by error propagation", Tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science (1985).

Sainath, T. N. and C. Parada, "Convolutional neural networks for small-footprint keyword spotting", in "Sixteenth Annual Conference of the International Speech Communication Association", (2015).

SCTK, "The NIST scoring toolkit", URL `https://github.com/usnistgov/SCTK` (2008).

Shafique, M., T. Theocharides, C.-S. Bouganis, M. A. Hanif, F. Khalid, R. Hafiz and S. Rehman, "An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era", in "Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018", pp. 827–832 (IEEE, 2018).

Shah, M., J. Wang, D. Blaauw, D. Sylvester, H.-S. Kim and C. Chakrabarti, "A fixed-point neural network for keyword detection on resource constrained hardware", in "Signal Processing Systems (SiPS), 2015 IEEE Workshop on", pp. 1–6 (IEEE, 2015).

Shin, D., J. Lee, J. Lee and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks", in "Solid-State Circuits Conference (ISSCC), 2017 IEEE International", pp. 240–241 (IEEE, 2017).

Song, S., K. D. Miller and L. F. Abbott, "Competitive hebbian learning through spike-timing-dependent synaptic plasticity", Nature neuroscience **3**, 9, 919 (2000).

Srivastava, G., D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti and J.-s. Seo, "Joint optimization of quantization and structured sparsity for compressed deep neural networks", in "ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)", pp. 1393–1397 (IEEE, 2019).

Su, D., X. Wu and L. Xu, "Gmm-hmm acoustic model training by a two level procedure with gaussian components determined by automatic model selection", in "Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on", pp. 4890–4893 (IEEE, 2010).

Sze, V., Y. Chen, J. Emer, A. Suleiman and Z. Zhang, "Hardware for Machine Learning: Challenges and Opportunities", (2016).

Tosic, I. and P. Frossard, "Dictionary learning", IEEE Signal Processing Magazine **28**, 2, 27–38 (2011).

Tu, M., V. Berisha, M. Woolf, J.-s. Seo and Y. Cao, "Ranking the parameters of deep neural networks using the fisher information", in "Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)", pp. 2647–2651 (2016).

Venkataramani, S., A. Ranjan, K. Roy and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing", in "Proceedings of the 2014 international symposium on Low power electronics and design", pp. 27–32 (ACM, 2014).

Wan, L., M. Zeiler, S. Zhang, Y. Le Cun and R. Fergus, "Regularization of neural networks using dropconnect", in "International Conference on Machine Learning", pp. 1058–1066 (2013).

Wang, I.-T., Y.-C. Lin, Y.-F. Wang, C.-W. Hsu and T.-H. Hou, "3d synaptic architecture with ultralow sub-10 fj energy per spike for neuromorphic computation", in "Electron Devices Meeting (IEDM), 2014 IEEE International", pp. 28–5 (IEEE, 2014).

Wang, M., Z. Wang, J. Lu, J. Lin and Z. Wang, "E-lstm: An efficient hardware architecture for long short-term memory", IEEE Journal on Emerging and Selected Topics in Circuits and Systems (2019).

Wang, S., Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang and Y. Liang, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs", in "Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)", pp. 11–20 (2018).

Wang, Z., "Cmos adjustable schmitt triggers", IEEE Transactions on instrumentation and Measurement **40**, 3, 601–605 (1991).

Wen, W., Y. He, S. Rajbhandari, M. Zhang, W. Wang, F. Liu, B. Hu, Y. Chen and H. Li, "Learning intrinsic sparse structures within long short-term memory", arXiv preprint arXiv:1709.05027 (2017).

Wen, W., C. Wu, Y. Wang, Y. Chen and H. Li, "Learning structured sparsity in deep neural networks", in "Advances in Neural Information Processing Systems", pp. 2074–2082 (2016).

Werbos, P. J., "Backpropagation through time: what it does and how to do it", Proceedings of the IEEE **78**, 10, 1550–1560 (1990).

Wilpon, J., L. Miller and P. Modi, "Improvements and applications for key word recognition using hidden markov modeling techniques", in "Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on", pp. 309–312 (IEEE, 1991).

Wong, H.-S. P., H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen and M.-J. Tsai, "Metal–oxide rram", Proceedings of the IEEE **100**, 6, 1951–1970 (2012).

Xiong, W., J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu and G. Zweig, "The Microsoft 2016 Conversational Speech Recognition System", (2016).

Xiong, W., L. Wu, F. Alleva, J. Droppo, X. Huang and A. Stolcke, "The microsoft 2017 conversational speech recognition system", in "Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)", pp. 5934–5938 (2018).

Xu, C., X. Dong, N. P. Jouppi and Y. Xie, "Design implications of memristor-based rram cross-point structures", in "Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011", pp. 1–6 (IEEE, 2011).

Xu, Z., A. Mohanty, P.-Y. Chen, D. Kadetotad, B. Lin, J. Ye, S. Vrudhula, S. Yu, J.-s. Seo and Y. Cao, "Parallel programming of resistive cross-point array for synaptic plasticity", Procedia Computer Science **41**, 126–133 (2014).

Ye, S., T. Zhang, K. Zhang, J. Li, J. Xie, Y. Liang, S. Liu, X. Lin and Y. Wang, "A unified framework of dnn weight pruning and weight clustering/quantization using admm", arXiv preprint arXiv:1811.01907 (2018).

Yin, S., D. Kadetotad, B. Yan, C. Song, Y. Chen, C. Chakrabarti and J.-s. Seo, "Low-power neuromorphic speech recognition engine with coarse-grain sparsity", in "2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)", pp. 111–114 (IEEE, 2017a).

Yin, S., P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu and S. Wei, "A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications", in "VLSI Circuits, 2017 Symposium on", pp. C26–C27 (IEEE, 2017b).

Yin, S., G. Srivastava, S. K. Venkataramanaiah, C. Chakrabarti, V. Berisha and J. Seo, "Minimizing area and energy of deep learning hardware design using collective low precision and structured compression", in "Proceedings of the 51st Asilomar Conference on Signals, Systems, and Computers", pp. 1907–1911 (2017c).

Yin, S., G. Srivastava, S. K. Venkataramanaiah, C. Chakrabarti, V. Berisha and J. Seo, "Minimizing Area and Energy of Deep Learning Hardware Design Using Collective Low Precision and Structured Compression", (2018).

Yu, D. and L. Deng, *AUTOMATIC SPEECH RECOGNITION.* (Springer, 2016).

Yu, J., A. Lukefahr, D. Palframan, G. Dasika, R. Das and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism", in "Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)", pp. 548–560 (2017).

Yu, S., B. Gao, Z. Fang, H. Yu, J. Kang and H.-S. P. Wong, "A low energy oxide-based electronic synaptic device for neuromorphic visual systems with tolerance to device variation", Advanced Materials **25**, 12, 1774–1779 (2013).

Yuan, F., "A high-speed differential cmos schmitt trigger with regenerative current feedback and adjustable hysteresis", Analog Integrated Circuits and Signal Processing **63**, 1, 121–127 (2010).

Zhou, J., K.-H. Kim and W. Lu, "Crossbar rram arrays: Selector device requirements during read operation", IEEE Transactions on Electron Devices **61**, 5, 1369–1376 (2014).

Zhu, F., J. Pool, M. Andersch, J. Appleyard and F. Xie, "Sparse persistent RNNs: Squeezing large recurrent networks on-chip", arXiv preprint arXiv:1804.10223 (2018).

Zhu, M. and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression", arXiv preprint arXiv:1710.01878 (2017).

Zylberberg, J., J. T. Murphy and M. R. DeWeese, "A sparse coding model with synaptically local plasticity and spiking neurons can account for the diverse shapes of v1 simple cell receptive fields", PLoS computational biology **7**, 10, e1002250 (2011).

# VITA

EDUCATION

**Ph.D., Electrical Engineering** *08/2013 - 05/2019*

Arizona State University, Tempe, Arizona, USA

**B.E., Electronics & Communication Engineering** *08/2009 - 5/2013*

M. S. Ramaiah Institute of Technology, Bengaluru, Karnataka, India